25
The Wavelet Transform

The concept of a transform was introduced in Section 24.1 and the rest of Chapter 24 discusses orthogonal transforms. The transforms dealt with in this chapter are different and are referred to as subband transforms, because they partition an image into various bands or regions that contain different features of the image.

We start with the simple concept of a signal. In mathematics, a function is normally denoted by $y = f(x)$. For our purposes, a signal is simply a function $x(t)$. The independent variable is denoted by t because signals that are of practical interest are functions of the time. When a signal is displayed graphically, we see its amplitude at any time and we can also measure its frequency at any point in time. We therefore say that the graphical representation of a signal is its time-amplitude representation or that it represents the *time domain* of the signal.

Many problems in science and engineering depend on how the frequency of a signal varies with time. In such a problem, the most important information of the signal is hidden in its frequency content. Such problems may be easier to solve when the signal is represented in the *frequency domain*.

The concept of frequency domain originated with Joseph Fourier who developed it in the early 1800s as part of his work on heat transfer. The Fourier transform changes the representation of a function from the time domain to the frequency domain. It has many applications and has been the subject of much research and experimentation.

As an example, consider the signal $x(t) = \cos(20t\pi) + \cos(50t\pi) + \cos(100t\pi) + \cos(200t\pi)$. The cosine function is periodic with a period of 2π. Thus, our signal contains the four periodic frequencies 10, 25, 50, and 100 (measured in units of 2π). Figure 25.1 illustrates the two domains of this signal. The time domain is a complex, infinite wave, but the frequency domain consists of only four peaks, indicating that this signal consists of four frequencies.

Our signal is relatively simple. In particular, its frequencies are always the same and do not vary over time. Such a signal is termed stationary. In general, signals are not stationary; they consist of frequencies that vary with time. The Fourier transform,

D. Salomon, *The Computer Graphics Manual*, Texts in Computer Science,
DOI 10.1007/978-0-85729-886-7_25, © Springer-Verlag London Limited 2011

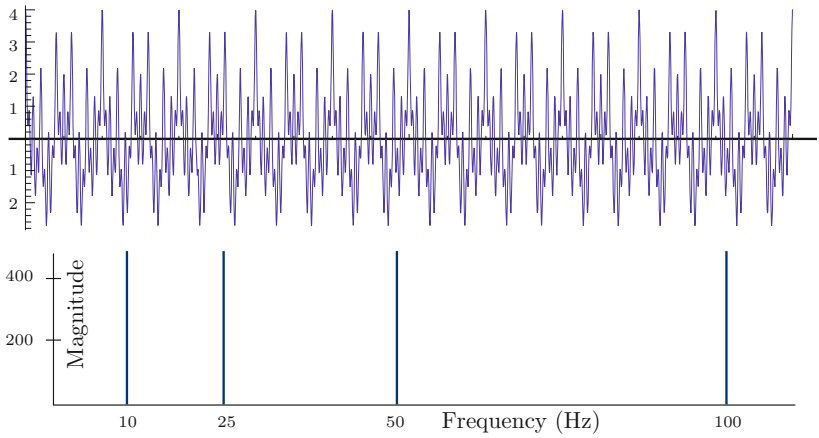


Figure 25.1: Time and Frequency Domains.

however, cannot tell the particular frequencies of a signal at any given time. It tells us what frequencies are included in the signal, but not when (at what values of t) each is present. We say that this transform has only frequency resolution but no time resolution.

The wavelet transform has been developed in the last few decades specifically to overcome this deficiency of the Fourier transform. The wavelet transform can tell what frequencies make up any part of a given signal. The complete details of this transform are outside the scope of this book, and this chapter deals only with those aspects of the wavelet transform that are relevant to the compression of images.

In practice, the wavelet transform is applied to data such as digitized audio and images. Such data consists of of individual numbers, and is therefore discrete, in contrast to mathematical signals, which are normally continuous. This is why the discrete, and not the continuous, wavelet transform is employed in the compression of images. The general discrete wavelet transform is described in Section 25.3

This chapter starts with the Haar transform, the simplest wavelet transform. It then describes how this simple transform can be extended and improved by means of filter banks. The discrete wavelet transform is then introduced, and the technique of compressing images by means of the wavelet transform is illustrated by the SPIHT algorithm.

25.1 The Haar Transform

The Haar transform is the simplest wavelet transform. It is presented here first informally, by means of an example, and then formally (in a short section that may be skipped by non-mathematically-savvy readers)

25.1.1 An Illustrating Example

This example is one dimensional. We consider a row of pixels, i.e., a one-dimensional array of n values. For simplicity we assume that n is a power of 2. (We use this assumption throughout this chapter, but there is no loss of generality. If n has a different value, the data can be extended by appending zeros. After decompression, the extra zeros are removed.) Consider the array of eight values $(1, 2, 3, 4, 5, 6, 7, 8)$. We first compute the four averages $(1 + 2)/2 = 3/2$, $(3 + 4)/2 = 7/2$, $(5 + 6)/2 = 11/2$, and $(7+8)/2 = 15/2$. It is impossible to reconstruct the original eight values from these four averages, so we also compute the four differences $(1-2)/2 = -1/2$, $(3-4)/2 = -1/2$, $(5-6)/2 = -1/2$, and $(7-8)/2 = -1/2$. These differences are called *detail coefficients*, and in this section the terms "difference" and "detail" are used interchangeably. We can think of the averages as a coarse resolution representation of the original image, and of the details as the data needed to reconstruct the original image from this coarse resolution. If the pixels of the image are correlated, the coarse representation will resemble the original pixels, while the details will be small. This explains why the Haar wavelet compression of images uses averages and details.

> Prolonged, lugubrious stretches of Sunday afternoon in a university town could be mitigated by attending Sillery's tea parties, to which anyone might drop in after half-past three. Action of some law of averages always regulated numbers at these gatherings to something between four and eight persons, mostly undergraduates, though an occasional don was not unknown.
>
> —Anthony Powell, *A Question of Upbringing* (1951).

It is easy to see that the array $(3/2, 7/2, 11/2, 15/2, -1/2, -1/2, -1/2, -1/2)$ made of the four averages and four differences can be used to reconstruct the original eight values. This array has eight values, but its last four components, the differences, tend to be small numbers, which helps in compression. Encouraged by this, we repeat the process on the four averages, the large components of our array. They are transformed into two averages and two differences, yielding $(10/4, 26/4, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$. The next, and last, iteration of this process transforms the first two components of the new array into one average (the average of all eight components of the original array) and one difference $(36/8, -16/8, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$. The last array is the *Haar wavelet transform* of the original data items.

Because of the differences, the wavelet transform tends to have numbers smaller than the original pixel values, so it is easier to compress using RLE, perhaps combined with move-to-front [Salomon 09] and Huffman coding. Lossy compression can be obtained if some of the smaller differences are quantized or even completely deleted (set to zero).

Before we continue, it is interesting (and also useful) to determine the complexity of this transform, i.e., the number of arithmetic operations as a function of the size of the data. In our example we needed $8+4+2 = 14$ operations (additions and subtractions), a number that can also be expressed as $14 = 2(8-1)$. In the general case, assume that we start with $N = 2^n$ data items. In the first iteration we need 2^n operations, in the second one we need 2^{n-1} operations, and so on, until the last iteration, where $2^{n-(n-1)} = 2^1$

operations are needed. Thus, the total number of operations is

$$\sum_{i=1}^{n} 2^i = \sum_{i=0}^{n} 2^i - 1 = \frac{1-2^{n+1}}{1-2} - 1 = 2^{n+1} - 2 = 2(2^n - 1) = 2(N-1).$$

The Haar wavelet transform of N data items can therefore be performed with $2(N-1)$ operations, so its complexity is $\mathcal{O}(N)$, an excellent result.

It is useful to associate with each iteration a quantity called *resolution*, which is defined as the number of remaining averages at the end of the iteration. The resolutions after each of the three iterations above are $4(=2^2)$, $2(=2^1)$, and $1(=2^0)$. Section 25.1.5 shows that each component of the wavelet transform should be normalized by dividing it by the square root of the resolution. (This is the *orthonormal Haar transform*, also discussed in Section 24.2.3.) Thus, our example wavelet transform becomes

$$\left(\frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}\right).$$

If the normalized wavelet transform is used, it can be formally proved that ignoring the smallest differences is the best choice for lossy wavelet compression, since it causes the smallest loss of image information.

The two procedures of Figure 25.2 illustrate how the normalized wavelet transform of an array of n components (where n is a power of 2) can be computed. Reconstructing the original array from the normalized wavelet transform is illustrated by the pair of procedures of Figure 25.3.

These procedures seem at first different from the averages and differences discussed earlier. They don't compute averages, because they divide by $\sqrt{2}$ instead of by 2; the first procedure starts by dividing the entire array by $\sqrt{n}$, and the second procedure ends by doing the reverse. The final result, however, is the same as that shown above. Starting with array $(1,2,3,4,5,6,7,8)$, the three iterations of procedure `NWTcalc` result in the following arrays

$$\left(\frac{3}{\sqrt{2^4}}, \frac{7}{\sqrt{2^4}}, \frac{11}{\sqrt{2^4}}, \frac{15}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{10}{\sqrt{2^5}}, \frac{26}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{36}{\sqrt{2^6}}, \frac{-16}{\sqrt{2^6}}, \frac{-4}{\sqrt{2^5}}, \frac{-4}{\sqrt{2^5}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}, \frac{-1}{\sqrt{2^4}}\right),$$
$$\left(\frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}\right).$$

25.1.2 A Formal Description

The use of the Haar transform for image compression is described here from a practical point of view. We first show how this transform is applied to the compression of grayscale

```
procedure NWTcalc(a:array of real, n:int);
 comment n is the array size (a power of 2)
 a:=a/√n comment divide entire array
 j:=n;
 while j≥ 2 do
  NWTstep(a, j);
  j:=j/2;
 endwhile;
end;

procedure NWTstep(a:array of real, j:int);
 for i=1 to j/2 do
  b[i]:=(a[2i-1]+a[2i])/√2;
  b[j/2+i]:=(a[2i-1]-a[2i])/√2;
 endfor;
 a:=b; comment move entire array
end;
```

Figure 25.2: Computing the Normalized Wavelet Transform.

```
procedure NWTreconst(a:array of real, n:int);
 j:=2;
 while j≤n do
  NWTRstep(a, j);
  j:=2j;
 endwhile
 a:=a√n; comment multiply entire array
end;

procedure NWTRstep(a:array of real, j:int);
 for i=1 to j/2 do
  b[2i-1]:=(a[i]+a[j/2+i])/√2;
  b[2i]:=(a[i]-a[j/2+i])/√2;
 endfor;
 a:=b; comment move entire array
end;
```

Figure 25.3: Restoring from a Normalized Wavelet Transform.

> We spake no word,
> Tho' each I ween did hear the other's soul.
> Not a wavelet stirred,
> And yet we heard
> The loneliest music of the weariest waves
> That ever roll.
> —Abram J. Ryan, *Poems*.

images, then show how this method can be extended to color images. One reference for Haar transform is [Stollnitz et al. 96].

The Haar transform uses a scale function $\phi(t)$ and a wavelet $\psi(t)$, both shown in Figure 25.4a, to represent a large number of functions $f(t)$. The representation is the infinite sum

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=0}^{\infty} d_{j,k} \psi(2^j t - k),$$

where c_k and $d_{j,k}$ are coefficients to be calculated.

The basic scale function $\phi(t)$ is the unit pulse

$$\phi(t) = \begin{cases} 1, & 0 \le t < 1, \\ 0, & \text{otherwise.} \end{cases}$$

The function $\phi(t-k)$ is a copy of $\phi(t)$, shifted k units to the right. Similarly, $\phi(2t-k)$ is a copy of $\phi(t-k)$ scaled to half the width of $\phi(t-k)$. The shifted copies are used to approximate $f(t)$ at different times t. The scaled copies are used to approximate $f(t)$ at higher resolutions. Figure 25.4b shows the functions $\phi(2^j t - k)$ for $j = 0, 1, 2$, and 3 and for $k = 0, 1, \ldots, 7$.

The basic Haar wavelet is the step function

$$\psi(t) = \begin{cases} 1, & 0 \le t < 0.5, \\ -1, & 0.5 \le t < 1. \end{cases}$$

From this we can see that the general Haar wavelet $\psi(2^j t - k)$ is a copy of $\psi(t)$ shifted k units to the right and scaled such that its total width is $1/2^j$.

◇ **Exercise 25.1:** Draw the four Haar wavelets $\psi(2^2 t - k)$ for $k = 0, 1, 2$, and 3.

Both $\phi(2^j t - k)$ and $\psi(2^j t - k)$ are nonzero in an interval of width $1/2^j$. This interval is their *support*. Since this interval tends to be short, we say that these functions have *compact support*.

We illustrate the basic transform on the simple step function

$$f(t) = \begin{cases} 5, & 0 \le t < 0.5, \\ 3, & 0.5 \le t < 1. \end{cases}$$

It is easy to see that $f(t) = 4\phi(t) + \psi(t)$. We say that the original steps $(5, 3)$ have been transformed to the (low resolution) average 4 and the (high resolution) detail 1. Using

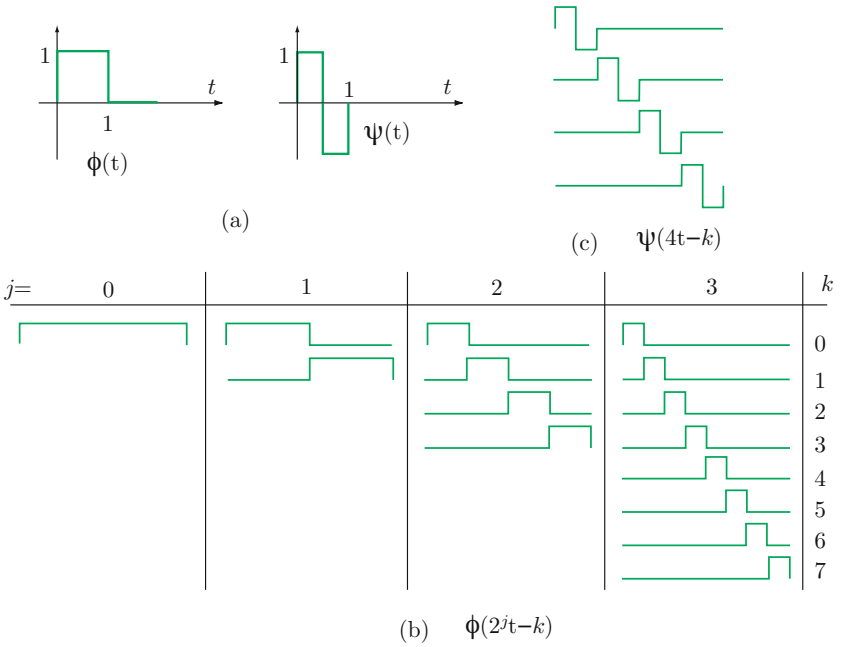


Figure 25.4: The Haar Basis Scale and Wavelet Functions.

matrix notation, this can be expressed (up to a factor of $\sqrt{2}$) as $(5,3)\mathbf{A}_2 = (4,1)$, where $\mathbf{A}_2$ is the order-2 Haar transform matrix of Equation (24.9).

Alfréd Haar (1885–1933)

Alfréd Haar was born in Budapest and received his higher mathematical training in Göttingen, where he later became a privatdozent. In 1912, he returned to Hungary and became a professor of mathematics first in Kolozsvár and then in Szeged, where he and his colleagues created a major mathematical center.

Haar is best remembered for his work on analysis on groups. In 1932 he introduced an invariant measure on locally compact groups, now called the Haar measure, which allows an analog of Lebesgue integrals to be defined on locally compact topological groups. Mathematical lore has it that John von Neumann tried to discourage Haar in this work because he felt certain that no such measure could exist. The following limerick celebrates Haar's achievement.

Said a mathematician named Haar,

"Von Neumann can't see very far.
He missed a great treasure—
They call it Haar measure—
Poor Johnny's just not up to par."

25.1.3 Applying the Haar Transform

Once the concept of a wavelet transform is grasped, it is easy to generalize it to a complete two-dimensional image. This can be done in several ways that are discussed in section 8.10 of [Salomon 09]. Here we show two such approaches, called the *standard decomposition* and the *pyramid decomposition*.

The former (Figure 25.6) starts by computing the wavelet transform of every row of the image. This results in a transformed image where the first column contains averages and all the other columns contain differences. The standard algorithm then computes the wavelet transform of every column. This results in one average value at the top-left corner, with the rest of the top row containing averages of differences, and with all other pixel values transformed into differences.

The latter method computes the wavelet transform of the image by alternating between rows and columns. The first step is to calculate averages and differences for all the rows (just one iteration, not the entire wavelet transform). This creates averages in the left half of the image and differences in the right half. The second step is to calculate averages and differences (just one iteration) for all the columns, which results in averages in the top-left quadrant of the image and differences elsewhere. Steps 3 and 4 operate on the rows and columns of that quadrant, resulting in averages concentrated in the top-left subquadrant. Pairs of steps are repeatedly executed on smaller and smaller subsquares, until only one average is left, at the top-left corner of the image, and all other pixel values have been reduced to differences. This process is summarized in Figure 25.7.

The transforms described in Section 24.1 are orthogonal. They transform the original pixels into a few large numbers and many small numbers. In contrast, wavelet transforms, such as the Haar transform, are *subband transforms*. They partition the image into regions such that one region contains large numbers (averages in the case of the Haar transform) and the other regions contain small numbers (differences). However, these regions, which are called subbands, are more than just sets of large and small numbers. They reflect different geometric artifacts of the image. To illustrate this important feature, we examine a small, mostly-uniform image with one vertical line and one horizontal line. Figure 25.5a shows an 8×8 image with pixel values of 12, except for a vertical line with pixel values of 14 and a horizontal line with pixel values of 16.

Figure 25.5b shows the results of applying the Haar transform once to the rows of the image. The right half of this figure (the differences) is mostly zeros, reflecting the uniform nature of the image. However, traces of the vertical line can easily be seen (the notation $\underline{2}$ indicates a negative difference). Figure 25.5c shows the results of applying the Haar transform once to the columns of Figure 25.5b. The upper-right subband now features traces of the vertical line, whereas the lower-left subband shows traces of the horizontal line. These subbands are denoted by HL and LH, respectively (Figures 25.7 and 25.29, although there is inconsistency in the use of this notation by various authors). The lower-right subband, denoted by HH, reflects diagonal image artifacts (which our

12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
16	16	16	16	14	16	16	16
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12

(a)

12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
16	16	15	16	0	0	2	0
12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0

(b)

12	12	13	12	0	0	2	0
12	12	13	12	0	0	2	0
14	14	14	14	0	0	0	0
12	12	13	12	0	0	2	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
4	4	2	4	0	0	4	0
0	0	0	0	0	0	0	0

(c)

Figure 25.5: An 8×8 Image and Its Subband Decomposition.

example image lacks). Most interesting is the upper-left subband, denoted by LL, that consists entirely of averages. This subband is a one-quarter version of the entire image, containing traces of both the vertical and the horizontal lines.

◇ **Exercise 25.2:** Construct a diagram similar to Figure 25.5 to show how subband HH reflects diagonal artifacts of the image.

(Artifact: A feature not naturally present, introduced during preparation or investigation.)

Figure 25.29 shows four levels of subbands, where level 1 contains the detailed features of the image (also referred to as the high-frequency or fine-resolution wavelet coefficients) and the top level, level 4, contains the coarse image features (low-frequency or coarse-resolution coefficients). It is clear that the lower levels can be quantized coarsely without much loss of important image information, while the higher levels should be quantized finely. The subband structure is the basis of all the image compression methods that use the wavelet transform.

Figure 25.8 shows typical results of the pyramid wavelet transform. The original image is shown at the top-left part of the figure. In order to illustrate how the pyramid transform works, this image consists only of horizontal, vertical, and slanted lines. The two halves of the top-right part of the figure show a left subband with the averages (this is similar to the entire image) and a right subband with the vertical details of the image. The bottom part of the figure features four subbands where the horizontal and diagonal details are also clear. The *Mathematica* code is also listed.

Section 24.1 discusses orthogonal transforms. An orthogonal linear transform is performed by computing the *inner product* of the data (pixel values or audio samples) with a set of *basis functions*. The result is a set of transform coefficients that can later be quantized or compressed with RLE, Huffman coding, or other methods.

The *subband transform*, on the other hand, is performed by computing a convolution of the data (Section 25.2) with a set of *bandpass filters*. Each resulting subband encodes a particular portion of the frequency content of the data.

As a reminder, the discrete inner product of the two vectors f_i and g_i is defined by

$$\langle f, g\rangle = \sum_i f_i\, g_i.$$

```
procedure StdCalc(a:array of real, n:int);
 comment array size is nxn (n = power of 2)
 for r=1 to n do NWTcalc(row r of a, n);
 endfor;
 for c=n to 1 do comment loop backwards
  NWTcalc(col c of a, n);
 endfor;
end;
procedure StdReconst(a:array of real, n:int);
 for c=n to 1 do comment loop backwards
  NWTreconst(col c of a, n);
 endfor;
 for r=1 to n do
  NWTreconst(row r of a, n);
 endfor;
end;
```

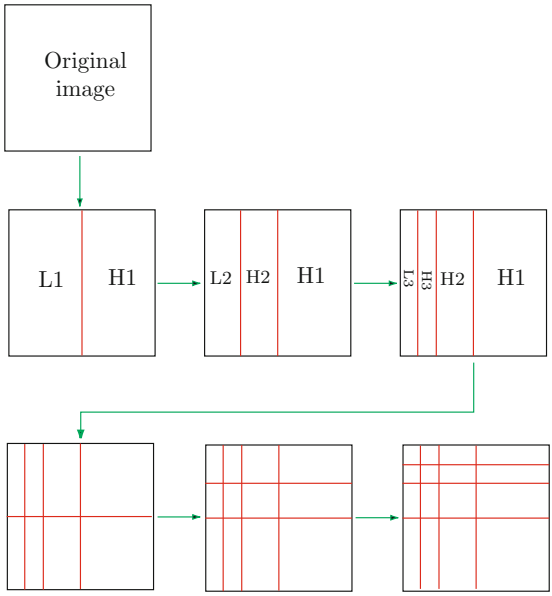


Figure 25.6: The Standard Image Wavelet Transform and Decomposition.

```
procedure NStdCalc(a:array of real, n:int);
 a:=a/√n comment divide entire array
 j:=n;
 while j≥ 2 do
  for r=1 to j do NWTstep(row r of a, j);
  endfor;
  for c=j to 1 do comment loop backwards
   NWTstep(col c of a, j);
  endfor;
  j:=j/2;
 endwhile;
end;
procedure NStdReconst(a:array of real, n:int);
 j:=2;
 while j≤n do
  for c=j to 1 do comment loop backwards
   NWTRstep(col c of a, j);
  endfor;
  for r=1 to j do
   NWTRstep(row r of a, j);
  endfor;
  j:=2j;
 endwhile
 a:=a√n; comment multiply entire array
end;
```

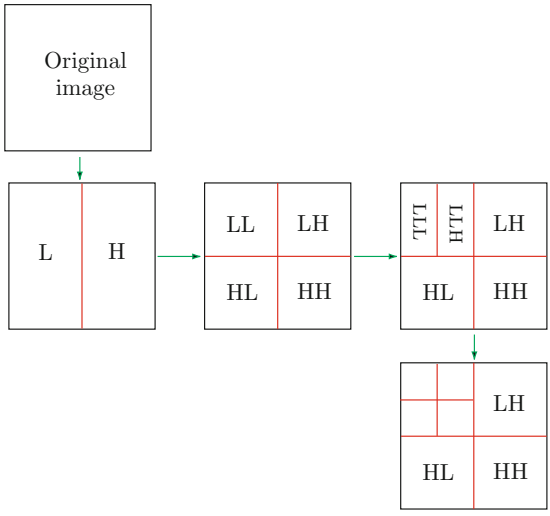
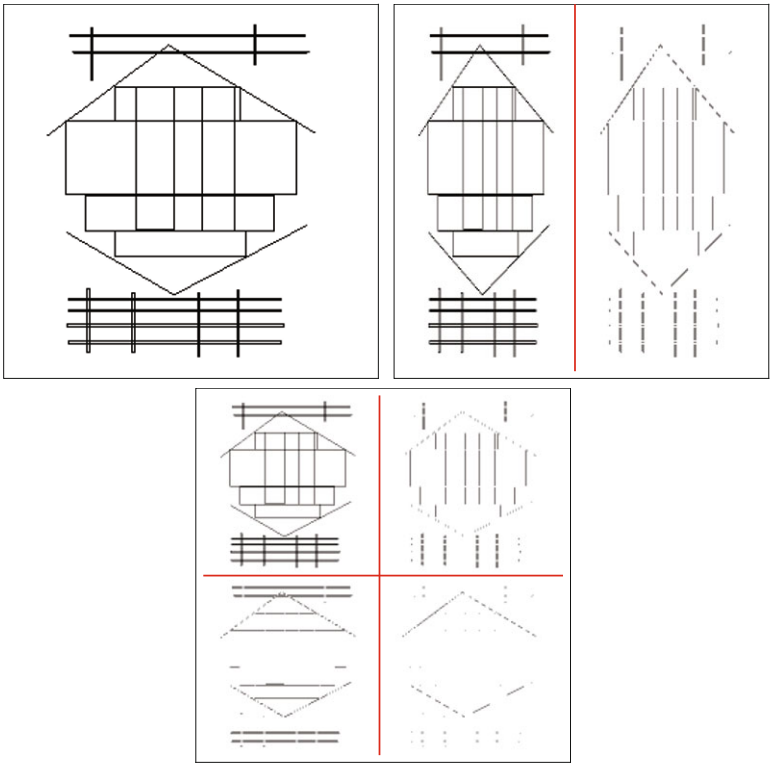


Figure 25.7: The Pyramid Image Wavelet Transform.



```
ar=Import["Design.raw", "Bit"];
stp=Partition[ar,256];
{row,col}=Dimensions[stp];
ArrayPlot[stp]
(* step 1, loop over columns and construct array ptp *)
ptp=Table[0,{i,1,row},{j,1,col}];(*Init ptp to zeros*)
mcol=Floor[col/2];
Do[ k=1;
Do[ptp[[i,k]]=(stp[[i,j]]+stp[[i,j+1]])/2;
 ptp[[i,mcol+k]]=(stp[[i,j]]-stp[[i,j+1]])/2; k=k+1,
 {j,1,col-1,2}], {i,1,row}]
ArrayPlot[ptp]
(* step 2, loop over the rows of ptp and construct array qtp *)
qtp=Table[0,{i,1,row},{j,1,col}];(*Init qtp to zeros*)
mrow=Floor[row/2];
Do[ k=1;
Do[qtp[[k,j]]=(ptp[[i,j]]+ptp[[i+1,j]])/2;
 qtp[[mrow+k,j]]=(ptp[[i,j]]-ptp[[i+1,j]])/2; k=k+1,
 {i,1,row-1,2}], {j,1,col}]
ArrayPlot[qtp]
```

Figure 25.8: A Pyramid Wavelet Decomposition.

The discrete convolution h is defined by Equation (25.1):

$$h_i = f \star g = \sum_j f_j \, g_{i-j}. \tag{25.1}$$

(Each element h_i of the discrete convolution h is the sum of products. It depends on i in the special way shown.)

Either method, standard or uniform, results in a transformed, although not yet compressed, image that has one average at the top-left corner and smaller numbers, differences, or averages of differences everywhere else. These numbers can be compressed using a combination of methods, such as RLE, move-to-front, and Huffman coding. If lossy compression is acceptable, some of the smallest differences can be quantized or even set to zeros, which creates run lengths of zeros, making the use of RLE even more attractive.

> Whiter foam than thine, O wave,
> Wavelet never wore,
> Stainless wave; and now you lave
> The far and stormless shore —
> Ever — ever — evermore!
> —Abram J. Ryan, *Poems.*

Color Images: So far we have assumed that each pixel is a single number (i.e., we have a single-component image, in which all pixels are shades of the same color, normally gray). Any compression method for single-component images can be extended to color (three-component) images by separating the three components, then transforming and compressing each individually. If the compression method is lossy, it makes sense to convert the three image components from their original color representation, which is normally RGB, to the YIQ color representation. The Y component of this representation is called *luminance*, and the I and Q (the chrominance) components are responsible for the color information (Chapter 21). The advantage of this color representation is that the human eye is most sensitive to Y and least sensitive to Q. A lossy method should therefore leave the Y component alone and delete some data from the I, and more data from the Q components, resulting in good compression and in a loss to which the eye is not that sensitive.

It is interesting to note that United States color television transmission also takes advantage of the YIQ representation. Signals are broadcast with bandwidths of 4 MHz for Y, 1.5 MHz for I, and only 0.6 MHz for Q.

25.1.4 Properties of the Haar Transform

The examples in this section illustrate some properties of the Haar transform, and of the discrete wavelet transform in general. Figure 25.10 shows a highly correlated 8×8 image and its Haar wavelet transform. Both the grayscale and numeric values of the pixels and the *transform coefficients* are shown. Because the original image is so correlated, the wavelet coefficients are small and there are many zeros.

◇ **Exercise 25.3:** A glance at Figure 25.10 suggests that the last sentence is wrong. The wavelet transform coefficients listed in the figure are very large compared with the pixel values of the original image. In fact, we know that the top-left Haar transform coefficient should be the average of all the image pixels. Since the pixels of our image have values that are (more or less) uniformly distributed in the interval $[0, 255]$, this average should be around 128, yet the top-left transform coefficient is 1,051. Explain this!

In a discrete wavelet transform, most of the wavelet coefficients are details (or differences). The details in the lower levels represent the fine details of the image. As we move higher in the subband level, we find details that correspond to coarser image features. Figure 25.11a illustrates this concept. It shows an image that is smooth on the left and has "activity" (i.e., adjacent pixels that tend to be different) on the right. Part (b) of the figure shows the wavelet transform of the image. Low levels (corresponding to fine details) have transform coefficients on the right, since this is where the image activity is located. High levels (coarse details) look similar but also have coefficients on the left side, because the image is not completely blank on the left.

The Haar transform is the simplest wavelet transform, but even this simple method illustrates the power of the wavelet transform. It turns out that the low levels of the discrete wavelet transform contain the unimportant image features, so quantizing or discarding these coefficients can lead to lossy compression that is both efficient and of high quality. Often, the image can be reconstructed from very few transform coefficients without any noticeable loss of quality. Figure 25.12a–c shows three reconstructions of the simple 8×8 image of Figure 25.10. They were obtained from only 32, 13, and 5 wavelet coefficients, respectively.

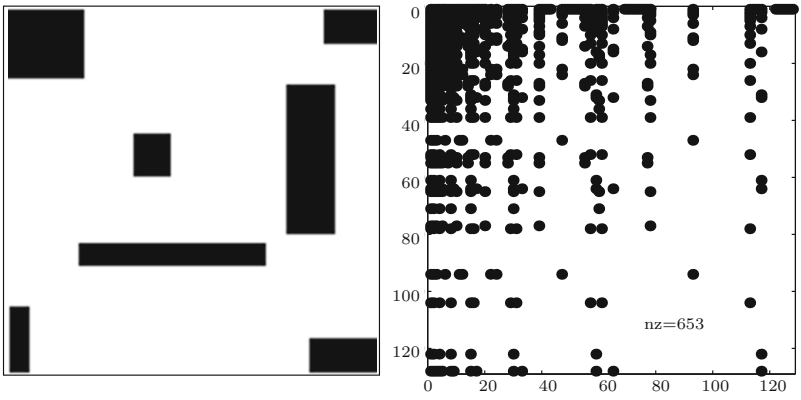
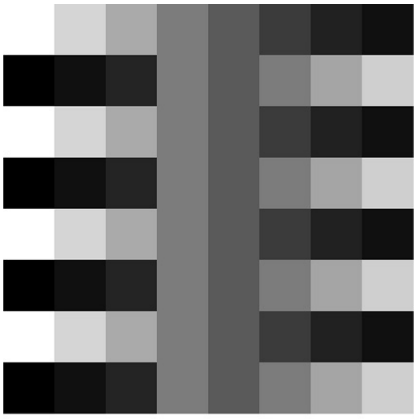
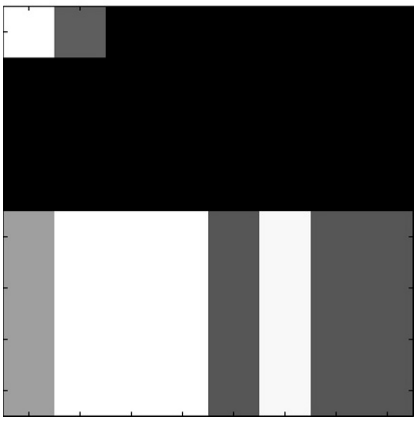


Figure 25.9: Reconstructing a 128×128 Simple Image from 4% of Its Coefficients.

Figure 25.9 is a similar example. It shows a bi-level image fully reconstructed from just 4% of its transform coefficients (653 coefficients out of 128×128).



255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223
255	224	192	159	127	95	63	32
0	32	64	159	127	159	191	223



1051	34.0	-44.5	-0.7	-1.0	-62	0	-1.0
0	0.0	0.0	0.0	0.0	0	0	0.0
0	0.0	0.0	0.0	0.0	0	0	0.0
0	0.0	0.0	0.0	0.0	0	0	0.0
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5
48	239.5	112.8	90.2	31.5	64	32	31.5

Figure 25.10: The Image that is Reconstructed in Figure 25.12 and Its Haar Transform.

(a) (b)

Figure 25.11: (a) A 128×128 Image with Activity on the Right. (b) Its Transform.

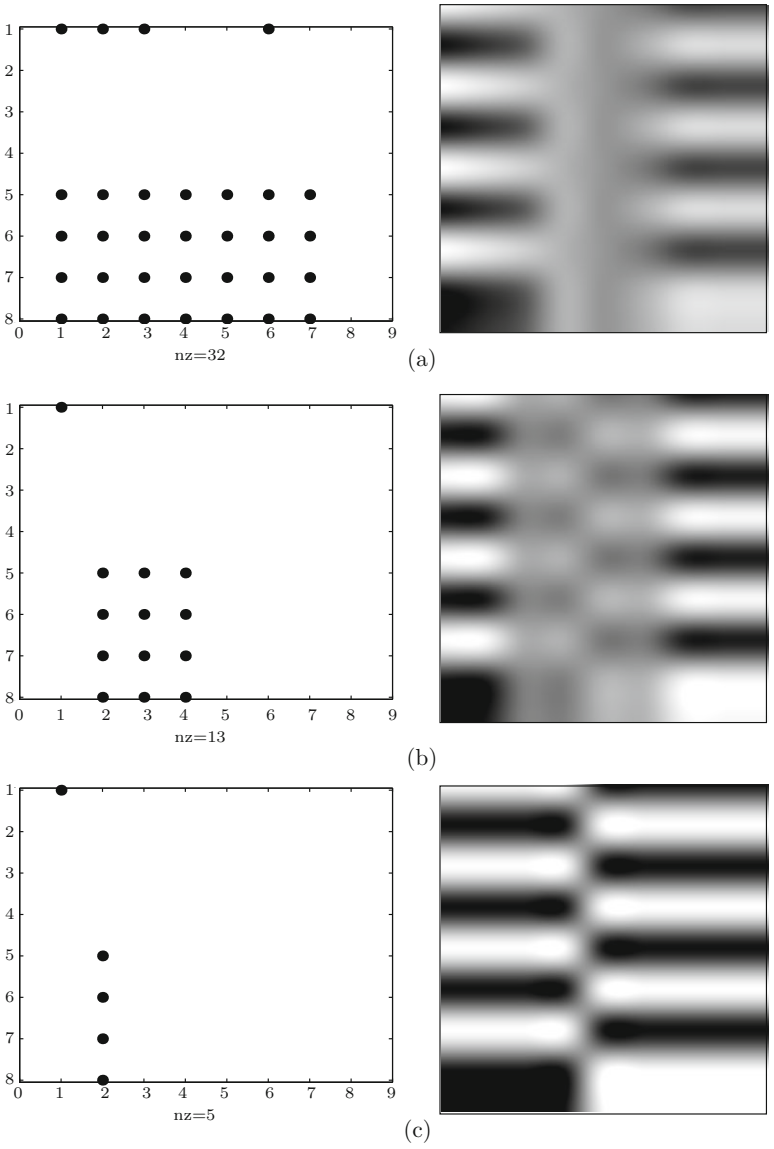


Figure 25.12: Three Lossy Reconstructions of an 8×8 Image.

Experimenting is the key to understanding these concepts. Proper mathematical software makes it easy to input images and experiment with various features of the discrete wavelet transform. In order to help the interested reader, Figure 25.13 lists a Matlab program that inputs an image, computes its Haar wavelet transform, discards a given percentage of the smallest transform coefficients, then computes the inverse transform to reconstruct the image.

Lossy wavelet image compression involves the discarding of coefficients, so the concept of *sparseness ratio* is defined to measure the amount of coefficients discarded. Sparseness is defined as the number of nonzero wavelet coefficients divided by the number of coefficients left after some are discarded. The higher the sparseness ratio, the fewer coefficients are left. Higher sparseness ratios lead to better compression but may result in poorly reconstructed images. The sparseness ratio is distantly related to *compression factor*, a compression measure defined in the Introduction.

The line "`filename='lena128'; dim=128;`" contains the image file name and the dimension of the image. The image files used by me were in raw form and contained just the grayscale values, each as a single byte. There is no header, and not even the image resolution (number of rows and columns) is included in the file. However, Matlab can read other types of files. The image is assumed to be square, and parameter "`dim`" should be a power of 2. The assignment "`thresh=`" specifies the percentage of transform coefficients to be deleted. This provides an easy way to experiment with lossy wavelet image compression.

File "`harmatt.m`" contains two functions that compute the Haar wavelet coefficients in a matrix form (Section 25.1.5).

(A technical note: A Matlab `m` file may include commands or a function but not both. It may, however, contain more than one function, provided that only the top function is invoked from outside the file. All the other functions must be called from within the file. In our case, function `harmatt(dim)` calls function `individ(n)`.)

◇ **Exercise 25.4:** Use the code of Figure 25.13 (or similar code) to compute the Haar transform of the *Lena* image (Figure 24.40) and reconstruct it three times by discarding more and more detail coefficients.

25.1.5 A Matrix Approach

The principle of the Haar transform is to compute averages and differences. It turns out that this can be done by means of matrix multiplication ([Mulcahy 96] and [Mulcahy 97]). As an example, we look at the top row of the simple 8×8 image of Figure 25.10. Anyone with a little experience with matrices can construct a matrix that when multiplied by this vector creates a vector with four averages and four differences. Matrix A_1 of Equation (25.2) does that and, when multiplied by the top row of pixels of Figure 25.10, generates $(239.5, 175.5, 111.0, 47.5, 15.5, 16.5, 16.0, 15.5)$. Similarly, matrices A_2 and A_3 perform the second and third steps of the transform, respectively. The results are shown

```
clear; % main program
filename='lena128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('file not found')
else img=fread(fid,[dim,dim])'; fclose(fid);
end
thresh=0.0;       % percent of transform coefficients deleted
figure(1), imagesc(img), colormap(gray), axis off, axis square
w=harmatt(dim);  % compute the Haar dim x dim transform matrix
timg=w*img*w';   % forward Haar transform
tsort=sort(abs(timg(:)));
tthresh=tsort(floor(max(thresh*dim*dim,1)));
cim=timg.*(abs(timg) > tthresh);
[i,j,s]=find(cim);
dimg=sparse(i,j,s,dim,dim);
% figure(2) displays the remaining transform coefficients
%figure(2), spy(dimg), colormap(gray), axis square
figure(2), image(dimg), colormap(gray), axis square
cimg=full(w'*sparse(dimg)*w);     % inverse Haar transform
density = nnz(dimg);
disp([num2str(100*thresh) '% of smallest coefficients deleted.'])
disp([num2str(density) ' coefficients remain out of ' ...
 num2str(dim) 'x' num2str(dim) '.'])
figure(3), imagesc(cimg), colormap(gray), axis off, axis square

File harmatt.m with two functions

function x = harmatt(dim)
num=log2(dim);
p = sparse(eye(dim)); q = p;
i=1;
while i<=dim/2;
 q(1:2*i,1:2*i) = sparse(individ(2*i));
 p=p*q; i=2*i;
end
x=sparse(p);

function f=individ(n)
x=[1, 1]/sqrt(2);
y=[1,-1]/sqrt(2);
while min(size(x)) < n/2
 x=[x, zeros(min(size(x)),max(size(x)));...
   zeros(min(size(x)),max(size(x))), x];
end
while min(size(y)) < n/2
 y=[y, zeros(min(size(y)),max(size(y)));...
   zeros(min(size(y)),max(size(y))), y];
end
f=[x;y];
```

Figure 25.13: Matlab Code for the Haar Transform of an Image.

in Equation (25.3):

$$A_1 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}, \quad A_1 \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 239.5 \\ 175.5 \\ 111.0 \\ 47.5 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}, \tag{25.2}$$

$$A_2 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad A_3 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$A_2 \begin{pmatrix} 239.5 \\ 175.5 \\ 111.0 \\ 47.5 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix} = \begin{pmatrix} 207.5 \\ 79.25 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix}, \quad A_3 \begin{pmatrix} 207.5 \\ 79.25 \\ 32.0 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16.0 \\ 15.5 \end{pmatrix} = \begin{pmatrix} 143.375 \\ 64.125 \\ 32. \\ 31.75 \\ 15.5 \\ 16.5 \\ 16. \\ 15.5 \end{pmatrix}. \tag{25.3}$$

Instead of calculating averages and differences, all we have to do is construct matrices A_1, A_2, and A_3, multiply them to obtain $W = A_1 A_2 A_3$, and apply W to all the columns of the image I by multiplying $W \cdot I$:

$$W \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{-1}{8} & \frac{-1}{8} & \frac{-1}{8} & \frac{-1}{8} \\ \frac{1}{4} & \frac{1}{4} & \frac{-1}{4} & \frac{-1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{-1}{4} & \frac{-1}{4} \\ \frac{1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{-1}{2} \end{pmatrix} \begin{pmatrix} 255 \\ 224 \\ 192 \\ 159 \\ 127 \\ 95 \\ 63 \\ 32 \end{pmatrix} = \begin{pmatrix} 143.375 \\ 64.125 \\ 32 \\ 31.75 \\ 15.5 \\ 16.5 \\ 16 \\ 15.5 \end{pmatrix}.$$

This, of course, is only half the job. In order to compute the complete transform, we still have to apply W to the rows of the product $W \cdot I$, and we do this by applying it to

the columns of the transpose $(W\cdot I)^T$, then transposing the result. Thus, the complete transform is (see line `timg=w*img*w'` in Figure 25.13)

$$I_{\rm tr} = \bigl(W(W\cdot I)^T\bigr)^T = W\cdot I\cdot W^T.$$

The inverse transform is performed by

$$W^{-1}(W^{-1}\cdot I_{\rm tr}^T)^T = W^{-1}\bigl(I_{\rm tr}\cdot(W^{-1})^T\bigr),$$

and this is where the normalized Haar transform (mentioned on Page 1150) becomes important. Instead of computing averages (quantities of the form $(d_i + d_{i+1})/2$) and differences [quantities of the form $(d_i - d_{i+1})/2$], it is better to use the quantities $(d_i + d_{i+1})/\sqrt{2}$ and $(d_i - d_{i+1})/\sqrt{2}$. This results in an *orthonormal* matrix W, and it is well known that the inverse of such a matrix is simply its transpose. Thus, we can write the inverse transform in the simple form $W^T\cdot I_{\rm tr}\cdot W$ (line `cimg=full(w'*sparse(dimg)*w)` in Figure 25.13).

In between the forward and inverse transforms, some transform coefficients may be quantized or deleted. Alternatively, matrix $I_{\rm tr}$ may be compressed by means of run length encoding and/or Huffman codes.

Function `individ(n)` of Figure 25.13 starts with a 2×2 Haar transform matrix (notice that it uses $\sqrt{2}$ instead of 2) and then uses it to construct as many individual matrices A_i as necessary. Function `harmatt(dim)` combines those individual matrices to form the final Haar matrix for an image of `dim` rows and `dim` columns.

◇ **Exercise 25.5:** Perform the calculation $W\cdot I\cdot W^T$ for the 8×8 image of Figure 25.10.

25.2 Filter Banks

This section uses the matrix approach to the Haar transform to introduce the reader to the idea of *filter banks*. We show how the Haar transform can be interpreted as a bank of two filters, a lowpass and a highpass. We explain the terms "filter," "lowpass," and "highpass" and show how the idea of filter banks leads naturally to the concept of *subband transform*. The Haar transform, of course, is the simplest wavelet transform, so it is used here to illustrate the new concepts. However, using it as a filter bank may not be very efficient. Most practical applications of wavelet filters employ more sophisticated sets of filter coefficients, but they are all based on the concept of filters and filter banks [Strang and Nguyen 96].

> And just like the wavelet that moans on the beach,
> And, sighing, sinks back to the sea,
> So my song—it just touches the rude shores of speech,
> And its music melts back into me.
>
> —Abram J. Ryan, *Poems*.

A *filter* is a linear operator defined in terms of its *filter coefficients* $h(0)$, $h(1)$, $h(2), \ldots$. It can be applied to an input vector x to produce an output vector y according to

$$y(n) = \sum_k h(k)x(n-k) = h \star x,$$

where the symbol $\star$ indicates a convolution. Notice that the limits of the sum above have not been stated explicitly. They depend on the sizes of vectors x and h. Since our independent variable is the time t, it is convenient to assume that the inputs (and, consequently, also the outputs) come at all times $t = \ldots, -2, -1, 0, 1, 2, \ldots$. Thus, we use the notation

$$x = (\ldots, a, b, c, d, e, \ldots),$$

where the central value c is the input at time zero [$c = x(0)$], values d and e are the inputs at times 1 and 2, respectively, and $b = x(-1)$ and $a = x(-2)$. In practice, the inputs are always finite, so the infinite vector x will have only a finite number of nonzero elements.

Deeper insight into the behavior of a linear filter can be gained by considering the simple input $x = (\ldots, 0, 0, 1, 0, 0, \ldots)$. This input is zero at all times except at $t = 0$. It is called a *unit pulse* or a *unit impulse*. Even though the limits of the sum in the convolution have not been specified, it is easy to see that for any n there is only one nonzero term in the sum, so $y(n) = h(n)x(0) = h(n)$. We say that the output $y(n) = h(n)$ at time n is the *response* at time n to the unit impulse $x(0) = 1$. Since the number of filter coefficients $h(i)$ is finite, this filter is a *finite impulse response* or FIR.

Figure 25.14 shows the basic idea of a filter bank. It shows an *analysis bank* consisting of two filters, a lowpass filter H_0 and a highpass filter H_1. The lowpass filter employs convolution to remove the high frequencies from the input signal x and let the low frequencies through. The highpass filter does the opposite. Together, they separate the input into *frequency bands*.

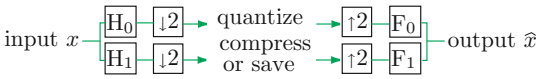


Figure 25.14: A Two-Channel Filter Bank.

The input x can be a one-dimensional signal (a vector of real numbers, which is what we assume in this section) or a two-dimensional signal, an image. The elements $x(n)$ of x are fed into the filters one by one, and each filter computes and outputs one number $y(n)$ in response to $x(n)$. The number of responses is therefore double the number of inputs (because we have two filters); a bad result, since we are interested in data compression. To correct this situation, each filter is followed by a *downsampling* process where the odd-numbered outputs are thrown away. This operation is also called *decimation* and is represented by the boxes marked "$\downarrow 2$". After decimation, the number of outputs from the two filters together equals the number of inputs.

Notice that the filter bank described here, followed by decimation, performs exactly the same calculations as matrix $W = A_1 A_2 A_3$ of Section 25.1.5. Filter banks are just a

more general way of looking at the Haar transform (or, in general, at the discrete wavelet transform). We look at this transform as a filtering operation, followed by decimation, and we can then try to find better filters.

The reason for having a bank of filters as opposed to just one filter is that several filters working together, with downsampling, can exhibit behavior that is impossible to obtain with just a single filter. The most important feature of a filter bank is its ability to reconstruct the input from the outputs H_0x and H_1x, even though each has been decimated.

Downsampling is not time invariant. After downsampling, the output is the even-numbered values $y(0)$, $y(2)$, $y(4)$,..., but if we delay the inputs by one time unit, the new outputs will be $y(-1)$, $y(1)$, $y(3)$,..., and these are different from and independent of the original outputs. These two sequences of signals are two phases of vector y.

The outputs of the analysis bank are called *subband coefficients*. They can be quantized (if lossy compression is acceptable), and they can be compressed by means of RLE, Huffman, arithmetic coding, or any other method. Eventually, they are fed into the *synthesis bank*, where they are first upsampled (by inserting zeros for each odd-numbered coefficient that was thrown away), then passed through the inverse filters F_0 and F_1, and finally combined to form a single output vector $\hat{x}$. The output of each analysis filter (after decimation) is

$$(\downarrow y) = (\ldots, y(-4), y(-2), y(0), y(2), y(4), \ldots).$$

Upsampling inserts zeros for the decimated values, so it converts the output vector above to

$$(\uparrow y) = (\ldots, y(-4), 0, y(-2), 0, y(0), 0, y(2), 0, y(4), 0, \ldots).$$

Downsampling causes loss of data. Upsampling alone cannot compensate for it, because it simply inserts zeros for the missing data. In order to achieve lossless reconstruction of the original signal x, the filters have to be designed such that they compensate for this loss of data. One feature that is commonly used in the design of good filters is *orthogonality*. Figure 25.15 shows a set of orthogonal filters of size 4. The filters of the set are orthogonal because their dot product is zero

$$(a, b, c, d) \cdot (d, -c, b, -a) = 0.$$

Notice how similar H_0 and F_0 are (and also H_1 and F_1). It still remains, of course, to choose actual values for the four *filter coefficients* a, b, c, and d. A full discussion of this is outside the scope of this book, but Section 25.2.1 illustrates some of the methods and rules used in practice to determine the values of various filter coefficients. An example is the Daubechies D4 filter, whose values are listed in Equation (25.7).

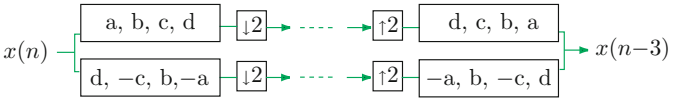


Figure 25.15: An Orthogonal Filter Bank with Four Filter Coefficients.

Simulating the operation of this filter manually shows that the reconstructed input is identical to the original input but lags three time units behind it.

A filter bank can also be *biorthogonal*, a less restricted type of filter. Figure 25.16 shows an example of such a set of filters that can reconstruct a signal exactly. Notice the similarity of H_0 and F_1 and also of H_1 and F_0.

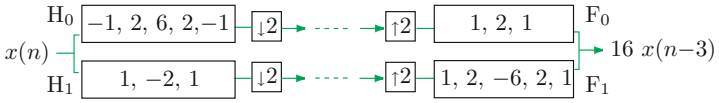


Figure 25.16: A Biorthogonal Filter Bank with Perfect Reconstruction.

We already know, from the discussion in Section 25.1, that the outputs of the low-pass filter H_0 are normally passed through the analysis filter several times, creating shorter and shorter outputs. This recursive process can be illustrated as a tree (Figure 25.17). Since each node of this tree produces half the number of outputs as its predecessor, the tree is called a *logarithmic tree*. Figure 25.17 shows how the scaling function $\phi(t)$ and the wavelet $\psi(t)$ are obtained at the limit of the logarithmic tree. This is the connection between the discrete wavelet transform (using filter banks) and the CWT.

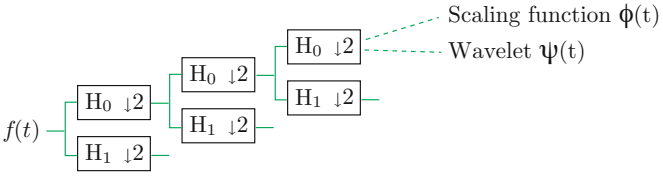


Figure 25.17: Scaling Function and Wavelet as Limits of a Logarithmic Tree.

As we "climb" up the logarithmic tree from level i to the next finer level $i+1$, we compute the new averages from the new, higher-resolution scaling functions $\phi(2^i t - k)$ and the new details from the new wavelets $\psi(2^i t - k)$

$$\begin{array}{r} \text{signal at level } i \text{ (averages)} \searrow \\ \qquad\qquad + \qquad \text{signal at level } i+1. \\ \text{details at level } i \text{ (differences)} \nearrow \end{array}$$

Each level of the tree corresponds to twice the frequency (or twice the resolution) of the preceding level, which is why the logarithmic tree is also called a *multiresolution tree*. Successive filtering through the tree separates lower and lower frequencies.

Those who do quantitative work with sound and music know that two tones at frequencies ω and 2ω sound virtually the same and differ only by pitch. The frequency interval between ω and 2ω is divided into 12 subintervals (the so-called *chromatic scale*),

but Western music has a tradition of favoring just eight of the twelve tones that result from this division (a *diatonic scale*, made up of seven notes, with the eighth note as the "octave"). This is why the basic frequency interval used in music is traditionally called an *octave*. We therefore say that adjacent levels of the multiresolution tree differ in an octave of frequencies.

In order to understand the meaning of *lowpass* and *highpass* we need to work in the frequency domain, where the convolution of two vectors is replaced by a multiplication of their Fourier transforms. The vector $x(n)$ is in the time domain, so its frequency domain equivalent is its discrete Fourier transform

$$X(\omega) \stackrel{\text{def}}{=} X(e^{i\omega}) = \sum_{-\infty}^{\infty} x(n)e^{-in\omega},$$

which is sometimes written in the z-domain,

$$X(z) = \sum_{-\infty}^{\infty} x(n)z^{-n},$$

where $z \stackrel{\text{def}}{=} e^{i\omega}$. The convolution by h in the time domain now becomes a multiplication by the function $H(\omega) = \sum h(n)e^{-in\omega}$ in the frequency domain, so we can express the output in the frequency domain by

$$Y(e^{i\omega}) = H(e^{i\omega})X(e^{i\omega}),$$

or, in reduced notation, $Y(\omega) = H(\omega)X(\omega)$, or, in the z-domain, $Y(z) = H(z)X(z)$. When all the inputs are $X(\omega) = 1$, the output at frequency ω is $Y(\omega) = H(\omega)$.

We can now understand the operation of the lowpass Haar filter. It works by averaging two consecutive inputs, so it produces the output

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n-1). \tag{25.4}$$

This is a convolution with only the two terms $k = 0$ and $k = 1$ in the sum. The filter coefficients are $h(0) = h(1) = 1/2$, and we can call the output a *moving average*, since each $y(n)$ depends on the current input and its predecessor. If the input is the unit impulse $x = (\ldots, 0, 0, 1, 0, 0, \ldots)$, then the output is $y(0) = y(1) = 1/2$, or $y = (\ldots, 0, 0, 1/2, 1/2, 0, \ldots)$. The output values are simply the filter coefficients as we saw earlier.

We can look at this averaging filter as the combination of an identity operator and a delay operator. The output produced by the identity operator equals the current input, while the output produced by the delay is the input one time unit earlier. Thus, we write

$$\text{averaging filter} = \frac{1}{2}(\text{identity}) + \frac{1}{2}(\text{delay}).$$

In matrix notation this can be expressed by

$$\begin{pmatrix} \cdots \\ y(-1) \\ y(0) \\ y(1) \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ & \frac{1}{2} & \frac{1}{2} & & \\ & & \frac{1}{2} & \frac{1}{2} & \\ & & & \cdots & \end{pmatrix} \begin{pmatrix} \cdots \\ x(-1) \\ x(0) \\ x(1) \\ \cdots \end{pmatrix}.$$

The 1/2 values on the main diagonal are copies of the weight of the identity operator. They all equal the $h(0)$ Haar filter coefficient. The 1/2 values on the diagonal below are copies of the weights of the delay operator. They all equal the $h(1)$ Haar filter coefficient. Thus, the matrix is a *constant diagonal* matrix (or a *banded* matrix). A wavelet filter that has a coefficient $h(3)$ would correspond to a matrix where this filter coefficient appears on the second diagonal below the main diagonal. The rule of matrix multiplication produces the familiar convolution

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots = \sum_k h(k)x(n-k).$$

Notice that the matrix is lower triangular. The upper diagonal, which would naturally correspond to the filter coefficients $h(-1)$, $h(-2)$,..., is zero. All filter coefficients with negative indices must be zero, since such coefficients lead to outputs that precede the inputs in time. In the real world, we are used to a cause preceding its effect, so our finite impulse response filters should also be *causal*.

Summary: A causal FIR filter with $N+1$ filter coefficients $h(0), h(1), \ldots, h(N)$ (a filter with $N+1$ "taps") has $h(i) = 0$ for all negative i and for $i > N$. When expressed in terms of a matrix, the matrix is lower triangular and banded. Such filters are commonly used and are important.

From the Dictionary

Tap (noun).

1. A cylindrical plug or stopper for closing an opening through which liquid is drawn, as in a cask; spigot.
2. A faucet or cock.
3. A connection made at an intermediate point on an electrical circuit or device.
4. An act or instance of wiretapping.

To illustrate the frequency response of a filter we select an input vector of the form

$$x(n) = e^{in\omega} = \cos(n\omega) + i\sin(n\omega), \text{ for } -\infty < n < \infty.$$

This is a complex function whose real and imaginary parts are a cosine and a sine, respectively, both with frequency ω. It is known that the Fourier transform of a pulse contains all the frequencies, but the Fourier transform of a sine wave has just one frequency. The

smallest frequency is $\omega = 0$, for which the vector becomes $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$. The highest frequency is $\omega = \pi$, where the same vector becomes $x = (\ldots, 1, -1, 1, -1, 1, \ldots)$. The special feature of this input is that the output vector $y(n)$ is a multiple of the input.

For the moving average, the output (filter response) is

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n-1) = \frac{1}{2}e^{in\omega} + \frac{1}{2}e^{i(n-1)\omega} = \left(\frac{1}{2} + \frac{1}{2}e^{-i\omega}\right)e^{in\omega} = H(\omega)x(n),$$

where $H(\omega) = (\frac{1}{2} + \frac{1}{2}e^{-i\omega})$ is the *frequency response function* of the filter. Since $H(0) = 1/2 + 1/2 = 1$, we see that the input $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$ is transformed to itself. Also, $H(\omega)$ for small values of ω generates output that is very similar to the input. This filter "lets" the low frequencies through, hence the name "lowpass filter." For $\omega = \pi$, the input is $x = (\ldots, 1, -1, 1, -1, 1, \ldots)$ and the output is all zeros (since the average of 1 and -1 is zero). This lowpass filter smooths out the high-frequency regions (the bumps) of the input signal.

Notice that we can write

$$H(\omega) = \left(\cos\frac{\omega}{2}\right)e^{i\omega/2}.$$

When we plot the magnitude $|H(\omega)| = \cos(\omega/2)$ of $H(\omega)$ (Figure 25.18a), it is easy to see that it has a maximum at $\omega = 0$ (the lowest frequency) and two minima at $\omega = \pm\pi$ (the highest frequencies).

The highpass filter uses differences to pick up the high frequencies in the input signal, and reduces or removes the smooth (low frequency) parts. In the case of the Haar transform, the highpass filter computes

$$y(n) = \frac{1}{2}x(n) - \frac{1}{2}x(n-1) = h \star x,$$

where the filter coefficients are $h(0) = 1/2$ and $h(1) = -1/2$, or

$$h = (\ldots, 0, 0, 1/2, -1/2, 0, \ldots).$$

In matrix notation this can be expressed by

$$\begin{pmatrix} \cdots \\ y(-1) \\ y(0) \\ y(1) \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots & & & & \\ -\frac{1}{2} & \frac{1}{2} & & & \\ & -\frac{1}{2} & \frac{1}{2} & & \\ & & -\frac{1}{2} & \frac{1}{2} & \\ & & & \cdots & \end{pmatrix} \begin{pmatrix} \cdots \\ x(-1) \\ x(0) \\ x(1) \\ \cdots \end{pmatrix}.$$

The main diagonal contains copies of $h(0)$, and the diagonal below contains $h(1)$. Using the identity and delay operator, this can also be written

$$\text{highpass filter} = \frac{1}{2}(\text{identity}) - \frac{1}{2}(\text{delay}).$$

Again selecting input $x(n) = e^{in\omega}$, it is easy to see that the output is

$$y(n) = \frac{1}{2}e^{in\omega} - \frac{1}{2}e^{i(n-1)\omega} = \left(\frac{1}{2} - \frac{1}{2}e^{-i\omega}\right) e^{-i\omega/2} = \sin(\omega/2)\, ie^{-i\omega/2}.$$

This time the highpass response function is

$$H_1(\omega) = \frac{1}{2} - \frac{1}{2}e^{-i\omega} = \frac{1}{2}\left(e^{i\omega/2} - e^{-i\omega/2}\right) e^{-i\omega/2} = \sin(\omega/2)\, e^{-i\omega/2}.$$

The magnitude is $|H_1(\omega)| = |\sin\left(\frac{\omega}{2}\right)|$. It is shown in Figure 25.18b, and it is obvious that it has a minimum for frequency zero and two maxima for large frequencies.

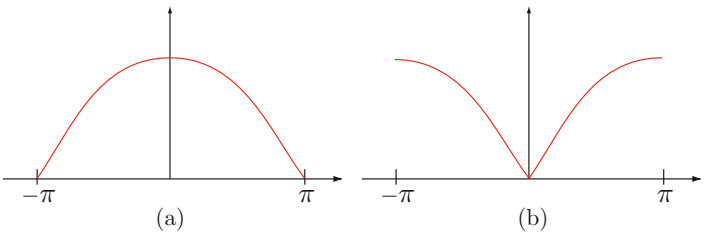


Figure 25.18: Magnitudes of (a) Lowpass and (b) Highpass Filters.

An important property of filter banks is that none of the individual filters are invertible, but the bank as a whole has to be designed such that the input signal could be perfectly reconstructed from the output in spite of the data loss caused by downsampling. It is easy to see, for example, that the constant signal $x = (\ldots, 1, 1, 1, 1, 1, \ldots)$ is transformed by the highpass filter H_1 to an output vector of all zeros. Obviously, there cannot exist an inverse filter H_1^{-1} that will be able to reconstruct the original input from a zero vector. The best that such an inverse transform can do is to use the zero vector to reconstruct another zero vector.

◇ **Exercise 25.6:** Show an example of an input vector x that is transformed by the lowpass filter H_0 to a vector of all zeros.

Summary: The discussion of filter banks in this section should be compared to the discussion of image transforms in Section 24.1. Even though both sections describe transforms, they differ in their approach, since they describe different classes of transforms. Each of the transforms described in Section 24.1 is based on a set of *orthogonal* basis functions (or orthogonal basis images), and is computed as an inner product of the input signal with the basis functions. The result is a set of transform coefficients that are subsequently compressed either losslessly (by RLE or some entropy encoder) or lossily (by quantization followed by entropy coding).

This section deals with subband transforms [Simoncelli and Adelson 90], a different type of transform that is computed by taking the *convolution* of the input signal with a

set of bandpass filters and decimating the results. Each decimated set of transform coefficients is a subband signal that encodes a specific range of the frequencies of the input. Reconstruction is done by upsampling, followed by computing the inverse transforms, and merging the resulting sets of outputs from the inverse filters.

The main advantage of subband transforms is that they isolate the different frequencies of the input signal, thereby making it possible for the user to precisely control the loss of data in each frequency range. In practice, such a transform decomposes an image into several subbands, corresponding to different image frequencies, and each subband can be quantized differently.

The main disadvantage of this type of transform is the introduction of artifacts, such as aliasing and ringing, into the reconstructed image, because of the downsampling. This is why the Haar transform is unsatisfactory, and most of the research in this field is concerned with finding better sets of filters.

Figure 25.19 illustrates a typical case of a general subband filter bank with N bandpass filters and three stages. Notice how the output of the lowpass filter H_0 of each stage is sent to the next stage for further decomposition, and how the combined output of the synthesis bank of a stage is sent to the top inverse filter of the synthesis bank of the preceding stage.

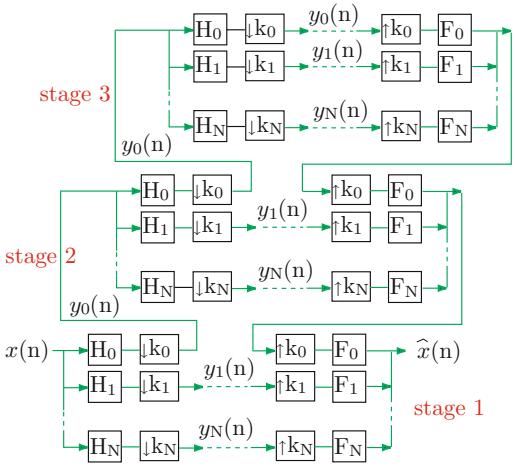


Figure 25.19: A General Filter Bank.

25.2.1 Deriving the Filter Coefficients

Once the basic operation of filter banks is understood, the natural question is, how are the filter coefficients derived? A full answer to this question is outside the scope of this book (see, for example, [Akansu and Haddad 92]), but this section provides a glimpse at the rules and methods used to figure out the values of various filter banks.

Given a set of two forward and two inverse N-tap filters H_0 and H_1, and F_0 and F_1 (where N is even), we denote their coefficients by

$$h_0 = \big(h_0(0), h_0(1), \ldots, h_0(N-1)\big), \quad h_1 = \big(h_1(0), h_1(1), \ldots, h_1(N-1)\big),$$
$$f_0 = \big(f_0(0), f_0(1), \ldots, f_0(N-1)\big), \quad f_1 = \big(f_1(0), f_1(1), \ldots, f_1(N-1)\big).$$

The four vectors h_0, h_1, f_0, and f_1 are the *impulse responses* of the four filters. The simplest set of conditions that these quantities have to satisfy is:

1. Normalization: Vector h_0 is normalized (i.e., its length is one unit).
2. Orthogonality: For any integer i that satisfies $1 \le i < N/2$, the vector formed by the first $2i$ elements of h_0 should be orthogonal to the vector formed by the last $2i$ elements of the same h_0.
3. Vector f_0 is the reverse of h_0.
4. Vector h_1 is a copy of f_0 where the signs of the odd-numbered elements (the first, third, etc.) are reversed. We can express this by saying that h_1 is computed by coordinate multiplication of h_1 and $(-1, 1, -1, 1, \ldots, -1, 1)$.
5. Vector f_1 is a copy of h_0 where the signs of the even-numbered elements (the second, fourth, etc.) are reversed. We can express this by saying that f_1 is computed by coordinate multiplication of h_0 and $(1, -1, 1, -1, \ldots, 1, -1)$.

For two-tap filters, rule 1 implies

$$h_0^2(0) + h_0^2(1) = 1. \tag{25.5}$$

Rule 2 is not applicable because $N = 2$, so $i < N/2$ implies $i < 1$. Rules 3–5 yield

$$f_0 = \big(h_0(1), h_0(0)\big), \quad h_1 = \big(-h_0(1), h_0(0)\big), \quad f_1 = \big(h_0(0), -h_0(1)\big).$$

It all depends on the values of $h_0(0)$ and $h_0(1)$, but the single Equation (25.5) is not enough to determine them. However, it is not difficult to see that the choice $h_0(0) = h_0(1) = 1/\sqrt{2}$ satisfies Equation (25.5).

For four-tap filters, rules 1 and 2 imply

$$h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) = 1, \quad h_0(0)h_0(2) + h_0(1)h_0(3) = 0, \tag{25.6}$$

and rules 3–5 yield

$$f_0 = \big(h_0(3), h_0(2), h_0(1), h_0(0)\big),$$
$$h_1 = \big(-h_0(3), h_0(2), -h_0(1), h_0(0)\big),$$
$$f_1 = \big(h_0(0), -h_0(1), h_0(2), -h_0(3)\big).$$

Again, Equation (25.6) is not enough to determine four unknowns, and other considerations (plus mathematical intuition) are needed to derive the four values. They are listed in Equation (25.7) (this is the Daubechies D4 filter).

◇ **Exercise 25.7:** Write the five conditions above for an eight-tap filter.

Determining the N filter coefficients for each of the four filters H_0, H_1, F_0, and F_1 depends on $h_0(0)$ through $h_0(N-1)$, so it requires N equations. However, in each of the cases above, rules 1 and 2 supply only $N/2$ equations. Other conditions have to be imposed and satisfied before the N quantities $h_0(0)$ through $h_0(N-1)$ can be determined. Here are some examples:

Lowpass H_0 filter: We want H_0 to be a lowpass filter, so it makes sense to require that the frequency response $H_0(\omega)$ be zero for the highest frequency $\omega = \pi$.

Minimum phase filter: This condition requires the zeros of the complex function $H_0(z)$ to lie on or inside the unit circle in the complex plane.

Controlled collinearity: The linearity of the phase response can be controlled by requiring that the sum

$$\sum_i \big(h_0(i) - h_0(N-1-i)\big)^2$$

be a minimum.

Other conditions are discussed in [Akansu and Haddad 92].

25.3 The DWT

Information that is produced and analyzed in real-life situations is discrete. It comes in the form of numbers, rather than as a continuous function. This is why the discrete, rather than the continuous, wavelet transform is used in practice ([Daubechies 88], [DeVore et al. 92], and [Vetterli and Kovacevic 95]). Section 8.5 of [Salomon 09] discusses the continuous wavelet transform (CWT) and shows that it is the integral of the product $f(t)\psi^*(\frac{t-b}{a})$, where a, the scale factor, and b, the time shift, can be any real numbers. The corresponding calculation for the discrete case (the DWT) involves a *convolution*, but experience shows that the quality of this type of transform depends heavily on two factors, the choice of scale factors and time shifts, and the choice of wavelet.

In practice, the DWT is computed with scale factors that are negative powers of 2 and time shifts that are nonnegative powers of 2. Figure 25.20 shows the so-called *dyadic lattice* that illustrates this particular choice. The wavelets used are those that generate orthonormal (or biorthogonal) wavelet bases.

The main thrust in wavelet research has therefore been the search for wavelet families that form orthogonal bases. Of those wavelets, the preferred ones are those that have compact support, because they allow for DWT computations with *finite impulse response* (FIR) filters.

The simplest way to describe the discrete wavelet transform is by means of matrix multiplication, along the lines developed in Section 25.1.5. The Haar transform depends on two *filter coefficients* c_0 and c_1, both with a value of $1/\sqrt{2} \approx 0.7071$. The smallest transform matrix that can be constructed in this case is $\left(\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right)/\sqrt{2}$. It is a 2×2 matrix, and it generates two transform coefficients, an average and a difference. (Notice that these are not exactly an average and a difference, because $\sqrt{2}$ is used instead of 2. Better names for them are *coarse detail* and *fine detail*, respectively.) In general, the DWT can

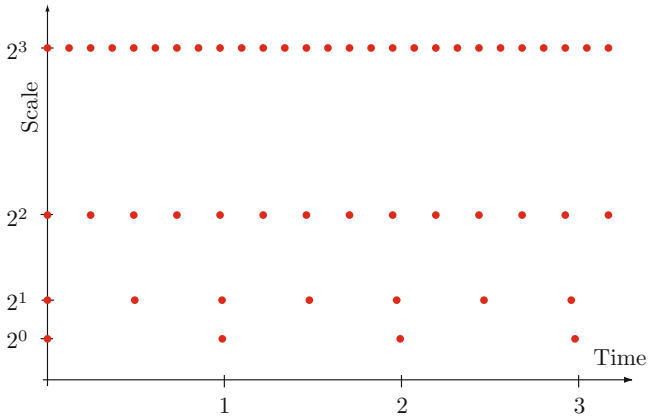


Figure 25.20: The Dyadic Lattice Showing the Relation Between Scale Factors and Time.

use any set of wavelet filters, but it is computed in the same way regardless of the particular filter used.

We start with one of the most popular wavelets, the Daubechies D4. As its name implies, it is based on four filter coefficients c_0, c_1, c_2, and c_3, whose values are listed in Equation (25.7). The transform matrix W is (compare with matrix A_1, Equation (25.2))

$$W = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & \dots & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & \dots & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & \dots & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & \dots & 0 \\ \vdots & \vdots & & & & \ddots & & \\ 0 & 0 & \dots & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & \dots & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & \dots & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & \dots & 0 & 0 & c_3 & -c_2 \end{pmatrix}.$$

When this matrix is applied to a column vector of data items $(x_1, x_2, \dots, x_n)$, its top row generates the weighted sum $s_1 = c_0x_1 + c_1x_2 + c_2x_3 + c_3x_4$, its third row generates the weighted sum $s_2 = c_0x_3 + c_1x_4 + c_2x_5 + c_3x_6$, and the other odd-numbered rows generate similar weighted sums s_i. Such sums are *convolutions* of the data vector x_i with the four filter coefficients. In the language of wavelets, each of them is called a *smooth coefficient*, and together they are called an H smoothing filter.

In a similar way, the second row of the matrix generates the quantity $d_1 = c_3x_1 - c_2x_2 + c_1x_3 - c_0x_4$, and the other even-numbered rows generate similar convolutions. Each d_i is called a *detail coefficient*, and together they are called a G filter. G is not a smoothing filter. In fact, the filter coefficients are chosen such that the G filter generates small values when the data items x_i are correlated. Together, H and G are

called *quadrature mirror filters* (QMF).

The discrete wavelet transform of an image can therefore be viewed as passing the original image through a QMF that consists of a pair of lowpass (H) and highpass (G) filters.

If W is an $n\times n$ matrix, it generates $n/2$ smooth coefficients s_i and $n/2$ detail coefficients d_i. The transposed matrix is

$$W^T=\begin{pmatrix}
c_0 & c_3 & 0 & 0 & \dots & & & & & c_2 & c_1\\
c_1 & -c_2 & 0 & 0 & \dots & & & & & c_3 & -c_0\\
c_2 & c_1 & c_0 & c_3 & \dots & & & & & 0 & 0\\
c_3 & -c_0 & c_1 & -c_2 & \dots & & & & & 0 & 0\\
 & & & & \ddots & & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & 0 & 0\\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & 0 & 0\\
 & & & & & & & c_2 & c_1 & c_0 & c_3\\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{pmatrix}.$$

It can be shown that in order for W to be orthonormal, the four coefficients have to satisfy the two relations $c_0^2+c_1^2+c_2^2+c_3^2=1$ and $c_2c_0+c_3c_1=0$. The other two equations used to calculate the four filter coefficients are $c_3-c_2+c_1-c_0=0$ and $0c_3-1c_2+2c_1-3c_0=0$. They represent the vanishing of the first two moments of the sequence $(c_3,-c_2,c_1,-c_0)$. The solutions are

$$\begin{aligned}
c_0&=(1+\sqrt{3})/(4\sqrt{2})\approx 0.48296, \quad c_1=(3+\sqrt{3})/(4\sqrt{2})\approx 0.8365,\\
c_2&=(3-\sqrt{3})/(4\sqrt{2})\approx 0.2241, \quad c_3=(1-\sqrt{3})/(4\sqrt{2})\approx -0.1294.
\end{aligned}\tag{25.7}$$

Using a transform matrix W is conceptually simple, but not very practical, since W should be of the same size as the image, which can be large. However, a look at W shows that it is very regular, so there is really no need to construct the full matrix. It is enough to have just the top row of W. In fact, it is enough to have just an array with the filter coefficients. Figure 25.21 lists Matlab code that performs this calculation. Function `fwt1(dat,coarse,filter)` takes a row vector `dat` of 2^n data items, and another array, `filter`, with filter coefficients. It then calculates the first `coarse` levels of the discrete wavelet transform.

◇ **Exercise 25.8:** Write similar code for the inverse one-dimensional discrete wavelet transform.

Plotting Functions: Wavelets are being used in many fields and have many applications, but the simple test of Figure 25.21 suggests another application, namely, plotting functions. Any graphics program or graphics software package has to include a routine to plot functions. It works by calculating the function at certain points and connecting the points with straight segments. In regions where the function has small curvature (it resembles a straight line) few points are needed, whereas in areas where the function has large curvature (it changes direction rapidly) more points are required. An ideal plotting routine should therefore be adaptive. It should select the points depending on the curvature of the function.

```
function wc1=fwt1(dat,coarse,filter)
%  The 1D Forward Wavelet Transform
%  dat must be a 1D row vector of size 2^n,
%  coarse is the coarsest level of the transform
%  (note that coarse should be <<n)
%  filter is an orthonormal quadrature mirror filter
%  whose length should be <2^(coarse+1)
n=length(dat); j=log2(n); wc1=zeros(1,n);
beta=dat;
for i=j-1:-1:coarse
  alfa=HiPass(beta,filter);
  wc1((2^(i)+1):(2^(i+1)))=alfa;
  beta=LoPass(beta,filter) ;
end
wc1(1:(2^coarse))=beta;

function d=HiPass(dt,filter) % highpass downsampling
d=iconv(mirror(filter),lshift(dt));
% iconv is matlab convolution tool
n=length(d);
d=d(1:2:(n-1));

function d=LoPass(dt,filter) % lowpass downsampling
d=aconv(filter,dt);
% aconv is matlab convolution tool with time-
% reversal of filter
n=length(d);
d=d(1:2:(n-1));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;
```

A simple test of `fwt1` is

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
```

which outputs

```
dat=
0.3827  0.7071  0.9239 1.0000 0.9239 0.7071 0.3827 0
-0.3827 -0.7071 -0.9239 -1.0000 -0.9239 -0.7071 -0.3827 0
wc=
1.1365 -1.1365 -1.5685 1.5685 -0.2271 -0.4239 0.2271 0.4239
-0.0281 -0.0818 -0.0876 -0.0421 0.0281 0.0818 0.0876 0.0421
```

Figure 25.21: Code for the One-Dimensional Forward Discrete Wavelet Transform.

The curvature, however, may not be easy to compute (it is essentially given by the second derivative of the function) which is why many plotting routines use instead the angle between consecutive segments. Figure 25.22 shows how a typical plotting routine works. It starts with a fixed number (say, 50) of points. This implies 49 straight segments connecting them. Before any of the segments is actually plotted, the routine measures the angles between consecutive segments. If an angle at point $\mathbf{P}_i$ is extreme (close to zero or close to 360°, as it is around points 4 and 10 in the figure), then more points are calculated between points $\mathbf{P}_{i-1}$ and $\mathbf{P}_{i+1}$; otherwise (if the angle is closer to 180°, as, for example, around points 5 and 9 in the figure), $\mathbf{P}_i$ is considered the only point necessary in that region.

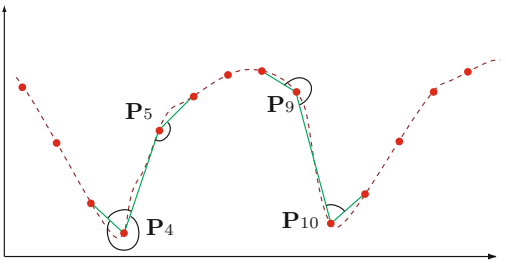


Figure 25.22: Using Angles Between Segments to Add More Points.

Better and faster results may be obtained using a discrete wavelet transform. The function is evaluated at n points (where n, a parameter, is large), and the values are collected in a vector v. A discrete wavelet transform of v is then computed, to produce n transform coefficients. The next step is to discard m of the smallest coefficients (where m is another parameter). We know, from the previous discussion, that the smallest coefficients represent small details of the function, so discarding them leaves the important details practically untouched. The inverse transform is then performed on the remaining $n - m$ transform coefficients, resulting in $n - m$ new points that are then connected with straight segments. The larger m, the fewer segments necessary, but the worse the fit.

Readers who take the trouble to read and understand functions `fwt1` and `iwt1` (Figures 25.21 and 25.23) may be interested in their two-dimensional equivalents, functions `fwt2` and `iwt2`, which are listed in Figures 25.24 and 25.25, respectively, with a simple test routine.

Table 25.26 lists the filter coefficients for some of the most common wavelets currently in use. Notice that each of those sets should still be normalized. Following are the main features of each set:

- The Daubechies family of filters maximize the smoothness of the father wavelet (the scaling function) by maximizing the rate of decay of its Fourier transform.
- The Haar wavelet can be considered the Daubechies filter of order 2. It is the oldest filter. It is simple to work with, but it does not produce best results, since it is not continuous.

```
function dat=iwt1(wc,coarse,filter)
% Inverse Discrete Wavelet Transform
dat=wc(1:2^coarse);
n=length(wc); j=log2(n);
for i=coarse:j-1
 dat=ILoPass(dat,filter)+ ...
  IHiPass(wc((2^(i)+1):(2^(i+1))),filter);
end

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% returns a vector of length 2*n with zeros
% placed between consecutive values
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

A simple test of `iwt1` is

```
n=16; t=(1:n)./n;
dat=sin(2*pi*t)
filt=[0.4830 0.8365 0.2241 -0.1294];
wc=fwt1(dat,1,filt)
rec=iwt1(wc,1,filt)
```

Figure 25.23: Code for the One-Dimensional Inverse Discrete Wavelet Transform.

```
function wc=fwt2(dat,coarse,filter)
%  The 2D Forward Wavelet Transform
%  dat must be a 2D matrix of size (2^n:2^n),
%  "coarse" is the coarsest level of the transform
%  (note that coarse should be <<n)
%  filter is an orthonormal qmf of length<2^(coarse+1)
q=size(dat); n = q(1); j=log2(n);
if q(1)~=q(2), disp('Nonsquare image!'), end;
wc = dat; nc = n;
for i=j-1:-1:coarse,
 top = (nc/2+1):nc; bot = 1:(nc/2);
 for ic=1:nc,
  row = wc(ic,1:nc);
  wc(ic,bot)=LoPass(row,filter);
  wc(ic,top)=HiPass(row,filter);
 end
 for ir=1:nc,
  row = wc(1:nc,ir)';
  wc(top,ir)=HiPass(row,filter)';
  wc(bot,ir)=LoPass(row,filter)';
 end
nc = nc/2;
end

function d=HiPass(dt,filter) % highpass downsampling
d=iconv(mirror(filter),lshift(dt));
% iconv is matlab convolution tool
n=length(d);
d=d(1:2:(n-1));

function d=LoPass(dt,filter) % lowpass downsampling
d=aconv(filter,dt);
% aconv is matlab convolution tool with time-
% reversal of filter
n=length(d);
d=d(1:2:(n-1));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;
```

A simple test of `fwt2` and `iwt2` is

```
filename='house128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('file not found')
 else img=fread(fid,[dim,dim])'; fclose(fid);
end
filt=[0.4830 0.8365 0.2241 -0.1294];
fwim=fwt2(img,4,filt);
figure(1), imagesc(fwim), axis off, axis square
rec=iwt2(fwim,4,filt);
figure(2), imagesc(rec), axis off, axis square
```

Figure 25.24: Code for the Two-Dimensional Forward Discrete Wavelet Transform.

```
function dat=iwt2(wc,coarse,filter)
% Inverse Discrete 2D Wavelet Transform
n=length(wc); j=log2(n);
dat=wc;
nc=2^(coarse+1);
for i=coarse:j-1,
 top=(nc/2+1):nc; bot=1:(nc/2); all=1:nc;
 for ic=1:nc,
  dat(all,ic)=ILoPass(dat(bot,ic)',filter)'  ...
   +IHiPass(dat(top,ic)',filter)';
 end % ic
 for ir=1:nc,
  dat(ir,all)=ILoPass(dat(ir,bot),filter)  ...
   +IHiPass(dat(ir,top),filter);
 end % ir
nc=2*nc;
end % i

function f=ILoPass(dt,filter)
f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% returns a vector of length 2*n with zeros
% placed between consecutive values
n =length(dt)*2; f =zeros(1,n);
f(1:2:(n-1))=dt;
```

A simple test of `fwt2` and `iwt2` is

```
filename='house128'; dim=128;
fid=fopen(filename,'r');
if fid==-1 disp('file not found')
 else img=fread(fid,[dim,dim])'; fclose(fid);
end
filt=[0.4830 0.8365 0.2241 -0.1294];
fwim=fwt2(img,4,filt);
figure(1), imagesc(fwim), axis off, axis square
rec=iwt2(fwim,4,filt);
figure(2), imagesc(rec), axis off, axis square
```

Figure 25.25: Code for the Two-Dimensional Inverse Discrete Wavelet Transform.

.099305765374	.424215360813	.699825214057	.449718251149	-.110927598348	-.264497231446
.026900308804	.155538731877	-.017520746267	-.088543630623	.019679866044	.042916387274
-.017460408696	-.014365807969	.010040411845	.001484234782	-.002736031626	.000640485329

Beylkin

.038580777748	-.126969125396	-.077161555496	.607491641386	.745687558934	.226584265197

Coifman 1-tap

.016387336463	-.041464936782	-.067372554722	.386110066823	.812723635450	.417005184424
-.076488599078	-.059434418646	.023680171947	.005611434819	-.001823208871	-.000720549445

Coifman 2-tap

-.003793512864	.007782596426	.023452696142	-.065771911281	-.061123390003	.405176902410
.793777222626	.428483476378	-.071799821619	-.082301927106	.034555027573	.015880544864
-.009007976137	-.002574517688	.001117518771	.000466216960	-.000070983303	-.000034599773

Coifman 3-tap

.000892313668	-.001629492013	-.007346166328	.016068943964	.026682300156	-.081266699680
-.056077313316	.415308407030	.782238930920	.434386056491	-.066627474263	-.096220442034
.039334427123	.025082261845	-.015211731527	-.005658286686	.003751436157	.001266561929
-.000589020757	-.000259974552	.000062339034	.000031229876	-.000003259680	-.000001784985

Coifman 4-tap

-.000212080863	.000358589677	.002178236305	-.004159358782	-.010131117538	.023408156762
.028168029062	-.091920010549	-.052043163216	.421566206729	.774289603740	.437991626228
-.062035963906	-.105574208706	.041289208741	.032683574283	-.019761779012	-.009164231153
.006764185419	.002433373209	-.001662863769	-.000638131296	.000302259520	.000140541149
-.000041340484	-.000021315014	.000003734597	.000002063806	-.000000167408	-.000000095158

Coifman 5-tap

.482962913145	.836516303738	.224143868042	-.129409522551		

Daubechies 4-tap

.332670552950	.806891509311	.459877502118	-.135011020010	-.085441273882	.035226291882

Daubechies 6-tap

.230377813309	.714846570553	.630880767930	-.027983769417	-.187034811719	.030841381836
.032883011667	-.010597401785				

Daubechies 8-tap

.160102397974	.603829269797	.724308528438	.138428145901	-.242294887066	-.032244869585
.077571493840	-.006241490213	-.012580751999	.003335725285		

Daubechies 10-tap

.111540743350	.494623890398	.751133908021	.315250351709	-.226264693965	-.129766867567
.097501605587	.027522865530	-.031582039317	.000553842201	.004777257511	-.001077301085

Daubechies 12-tap

.077852054085	.396539319482	.729132090846	.469782287405	-.143906003929	-.224036184994
.071309219267	.080612609151	-.038029936935	-.016574541631	.012550998556	.000429577973
-.001801640704	.000353713800				

Daubechies 14-tap

.054415842243	.312871590914	.675630736297	.585354683654	-.015829105256	-.284015542962
.000472484574	.128747426620	-.017369301002	-.044088253931	.013981027917	.008746094047
-.004870352993	-.000391740373	.000675449406	-.000117476784		

Daubechies 16-tap

.038077947364	.243834674613	.604823123690	.657288078051	.133197385825	-.293273783279
-.096840783223	.148540749338	.030725681479	-.067632829061	.000250947115	.022361662124
-.004723204758	-.004281503682	.001847646883	.000230385764	-.000251963189	.000039347320

Daubechies 18-tap

.026670057901	.188176800078	.527201188932	.688459039454	.281172343661	-.249846424327
-.195946274377	.127369340336	.093057364604	-.071394147166	-.029457536822	.033212674059
.003606553567	-.010733175483	.001395351747	.001992405295	-.000685856695	-.000116466855
.000093588670	-.000013264203				

Daubechies 20-tap

Table 25.26: Filter Coefficients for Some Common Wavelets (Continues).

- The Beylkin filter places the roots of the frequency response function close to the Nyquist frequency (a term that's explained in texts on digital signal processing and is mentioned in Section 2.1) on the real axis.
- The Coifman filter (or "Coiflet") of order p (where p is a positive integer) gives both the mother and father wavelets $2p$ zero moments.
- Symmetric filters (symmlets) are the most symmetric compactly supported wavelets with a maximum number of zero moments.
- The Vaidyanathan filter does not satisfy any conditions on the moments but produces exact reconstruction. This filter is especially useful in speech compression.

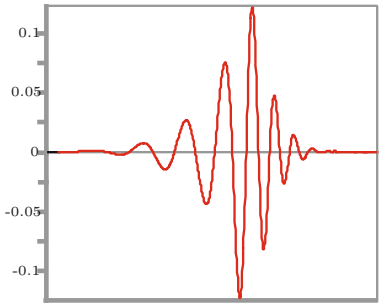
Figures 25.27 and 25.28 are diagrams of some of those wavelets.

-.107148901418	-.041910965125	.703739068656	1.136658243408	.421234534204	-.140317624179
-.017824701442	.045570345896				
		Symmlet 4-tap			
.038654795955	.041746864422	-.055344186117	.281990696854	1.023052966894	.896581648380
.023478923136	-.247951362613	-.029842499869	.027632152958		
		Symmlet 5-tap			
.021784700327	.004936612372	-.166863215412	-.068323121587	.694457972958	1.113892783926
.477904371333	-.102724969862	-.029783751299	.063250562660	.002499922093	-.011031867509
		Symmlet 6-tap			
.003792658534	-.001481225915	-.017870431651	.043155452582	.096014767936	-.070078291222
.024665659489	.758162601964	1.085782709814	.408183939725	-.198056706807	-.152463871896
.005671342686	.014521394762				
		Symmlet 7-tap			
.002672793393	-.000428394300	-.021145686528	.005386388754	.069490465911	-.038493521263
-.073462508761	.515398670374	1.099106630537	.680745347190	-.086653615406	-.202648655286
.010758611751	.044823623042	-.000766690896	-.004783458512		
		Symmlet 8-tap			
.001512487309	-.000669141509	-.014515578553	.012528896242	.087791251554	-.025786445930
-.270893783503	.049882830959	.873048407349	1.015259790832	.337658923602	-.077172161097
.000825140929	.042744433602	-.016303351226	-.018769396836	.000876502539	.001981193736
		Symmlet 9-tap			
.001089170447	.000135245020	-.012220642630	-.002072363923	.064950924579	.016418869426
-.225558972234	-.100240215031	.667071338154	1.088251530500	.542813011213	-.050256540092
-.045240772218	.070703567550	.008152816799	-.028786231926	-.001137535314	.006495728375
.000080661204	-.000649589896				
		Symmlet 10-tap			
-.000062906118	.000343631905	-.000453956620	-.000944897136	.002843834547	.000708137504
-.008839103409	.003153847056	.019687215010	-.014853448005	-.035470398607	.038742619293
.055892523691	-.077709750902	-.083928884366	.131971661417	.135084227129	-.194450471766
-.263494802488	.201612161775	.635601059872	.572797793211	.250184129505	.045799334111
		Vaidyanathan			

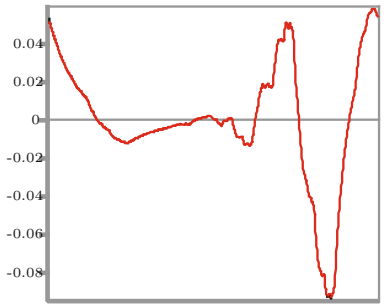
Table 25.26: Continued.

The Daubechies family of wavelets is a set of orthonormal, compactly supported functions where consecutive members are increasingly smoother. Some of them are shown in Figure 25.28. The term *compact support* means that these functions are zero (exactly zero, not just very small) outside a finite interval.

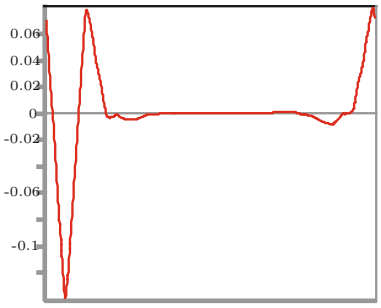
The Daubechies D4 wavelet is based on four coefficients, shown in Equation (25.7). The D6 wavelet is, similarly, based on six coefficients. They are determined by solving six equations, three of which represent orthogonality requirements and the other



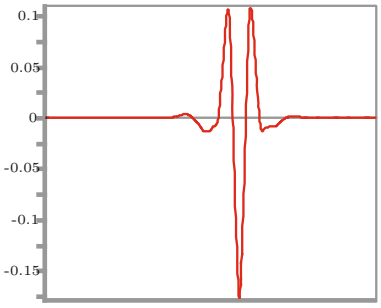
Beylkin 18



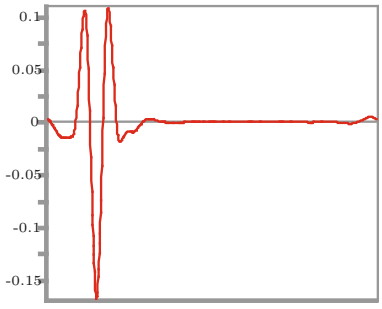
Coifman 6



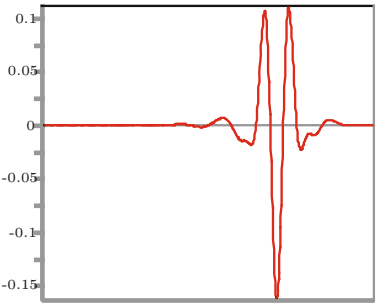
Coifman 12



Coifman 18

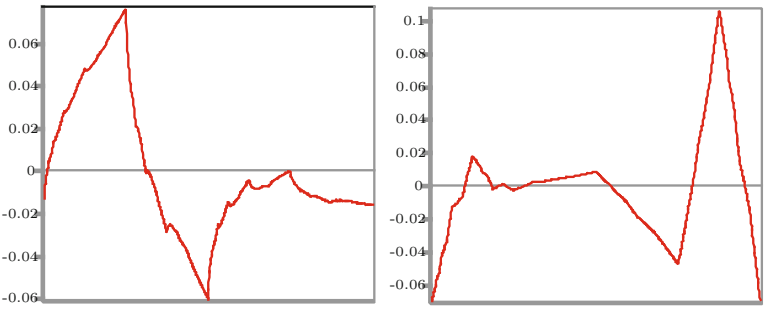


Coifman 24

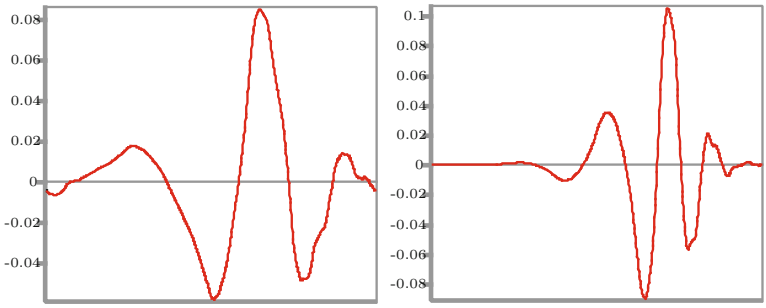


Coifman 30

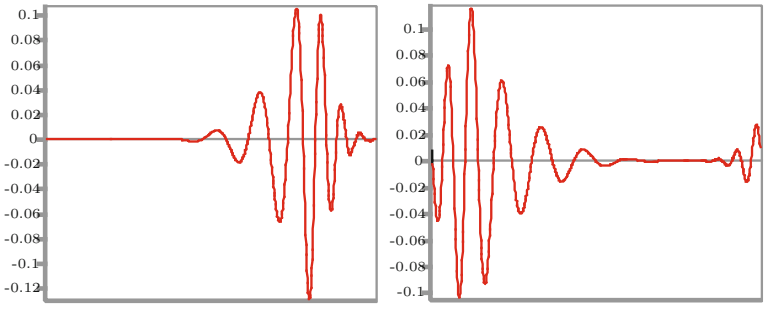
Figure 25.27: Examples of Common Wavelets.



Daubechies 4 Daubechies 6



Daubechies 8 Daubechies 10



Daubechies 20 Vaidyanathan 24

Figure 25.28: Examples of Common Wavelets.

three represent the vanishing of the first three moments. The result is shown in Equation (25.8):

$$\begin{aligned}
c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx .3326,\\
c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx .8068,\\
c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx .4598,\\
c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx -.1350,\\
c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx -.0854,\\
c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/(16\sqrt{2}) \approx .0352.
\end{aligned} \tag{25.8}$$

Each member of this family has two more coefficients than its predecessor and is smoother. The derivation of these functions is outside the scope of this book and can be found in [Daubechies 88]. They are derived recursively, do not have a closed form, and are nondifferentiable at infinitely many points (Page 1007). Truly unusual functions!

◇ **Exercise 25.9:** Use functions `fwt2` and `iwt2` of Figures 25.24 and 25.25 to blur an image. The idea is to compute the four-step subband transform of an image (thus ending up with 13 subbands), then set most of the transform coefficients to zero and heavily quantize some of the others. This, of course, results in a loss of image information, and in a nonperfectly reconstructed image. The aim of this exercise, however, is to have the inverse transform produce a *blurred image*. This illustrates an important property of the discrete wavelet transform, namely its ability to reconstruct images that degrade gracefully when more and more transform coefficients are zeroed or coarsely quantized. Other transforms, most notably the DCT, may introduce artifacts in the reconstructed image, but this property of the DWT makes it ideal for applications such as fingerprint compression (see section 8.18 of [Salomon 09]).

25.4 SPIHT

Section 25.1 shows how the Haar transform can be applied several times to an image, creating regions (or subbands) of averages and details. The Haar transform is simple, and better compression can be achieved by other wavelet filters. It seems that different wavelet filters produce different results depending on the image type, but it is currently not clear what filter is the best for any given image type. Regardless of the particular filter used, the image is decomposed into subbands, such that lower subbands correspond to higher image frequencies (they are the highpass levels) and higher subbands correspond to lower image frequencies (lowpass levels), where most of the image energy is concentrated (Figure 25.29). This is why we can expect the detail coefficients to get smaller as we move from high to low levels. Also, there are spatial similarities among the subbands (Figure 25.8b). An image feature, such as an edge, occupies the same spatial

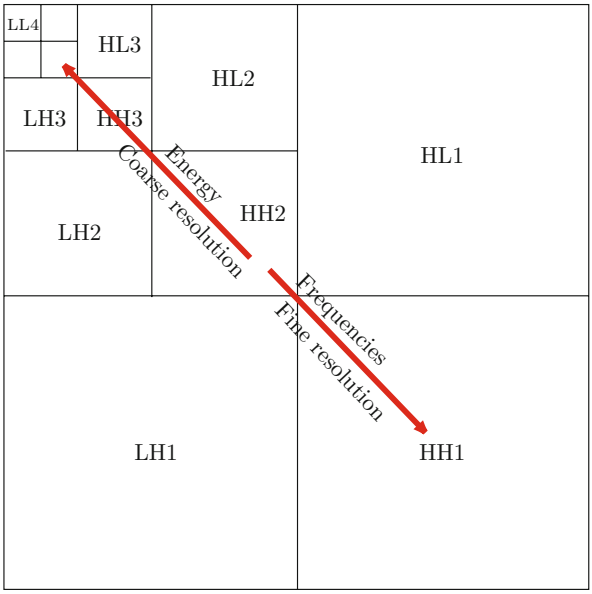


Figure 25.29: Subbands and Levels in Wavelet Decomposition.

position in each subband. These features of the wavelet decomposition are exploited by the SPIHT (set partitioning in hierarchical trees) method [Said and Pearlman 96].

SPIHT was designed for optimal progressive transmission, as well as for compression. One of the important features of SPIHT (perhaps a unique feature) is that at any point during the decoding of an image, the quality of the displayed image is the best that can be achieved for the number of bits input by the decoder up to that moment.

Another important SPIHT feature is its use of embedded coding. This feature is defined as follows: If an (embedded coding) encoder produces two files, a large one of size M and a small one of size m, then the smaller file is identical to the first m bits of the larger file.

The following example aptly illustrates the meaning of this definition. Suppose that three users are waiting for a certain compressed image, but they need different image qualities. The first user needs the quality contained in a 10 KB file. The image qualities required by the second and third users are contained in files of sizes 20 KB and 50 KB, respectively. Most lossy image compression methods would have to compress the same image three times, at different qualities, to generate three files with the right sizes. SPIHT, on the other hand, produces one file, and then three chunks—of lengths 10 KB, 20 KB, and 50 KB, all starting at the beginning of that file—can be sent to the three users, thereby satisfying their needs.

We start with a general description of SPIHT. We denote the pixels of the original image $\mathbf{p}$ by $p_{i,j}$. Any set $\mathbf{T}$ of wavelet filters can be used to transform the pixels to wavelet

coefficients (or transform coefficients) $c_{i,j}$. These coefficients constitute the transformed image $\mathbf{c}$. The transformation is denoted by $\mathbf{c} = \mathbf{T}(\mathbf{p})$. In a progressive transmission method, the decoder starts by setting the reconstruction image $\hat{\mathbf{c}}$ to zero. It then inputs (encoded) transform coefficients, decodes them, and uses them to generate an improved reconstruction image $\hat{\mathbf{c}}$, which in turn is used to produce a better image $\hat{\mathbf{p}}$. We can summarize this operation by $\hat{\mathbf{p}} = \mathbf{T}^{-1}(\hat{\mathbf{c}})$.

The main aim in progressive transmission is to transmit the most important image information first. This is the information that results in the largest reduction of the distortion (the difference between the original and the reconstructed images). SPIHT uses the mean squared error (MSE) distortion measure (Equation (23.2))

$$D_{\text{mse}}(\mathbf{p}-\hat{\mathbf{p}}) = \frac{|\mathbf{p}-\hat{\mathbf{p}}|^2}{N} = \frac{1}{N}\sum_i\sum_j (p_{i,j}-\hat{p}_{i,j})^2,$$

where N is the total number of pixels. An important consideration in the design of SPIHT is the fact that this measure is invariant to the wavelet transform, a feature that allows us to write

$$D_{\text{mse}}(\mathbf{p}-\hat{\mathbf{p}}) = D_{\text{mse}}(\mathbf{c}-\hat{\mathbf{c}}) = \frac{|\mathbf{p}-\hat{\mathbf{p}}|^2}{N} = \frac{1}{N}\sum_i\sum_j (c_{i,j}-\hat{c}_{i,j})^2. \tag{25.9}$$

Equation (25.9) shows that the MSE decreases by $|c_{i,j}|^2/N$ when the decoder receives the transform coefficient $c_{i,j}$ (we assume that the decoder receives the exact value of the coefficient, i.e., there is no loss of precision due to limitations imposed by computer arithmetic). It is now clear that the largest coefficients $c_{i,j}$ (largest in absolute value, regardless of their signs) contain the information that reduces the MSE distortion most, so a progressive encoder should send those coefficients first. This is an important principle of SPIHT.

Another principle is based on the observation that the most-significant bits of a binary integer whose value is close to maximum tend to be 1's. This suggests that the most significant bits contain the most important image information, and that they should be sent to the decoder first (or written first on the compressed stream).

The progressive transmission method used by SPIHT incorporates these two principles. SPIHT sorts the coefficients and transmits their most significant bits first. To simplify the description, we first assume that the sorting information is explicitly transmitted to the decoder; the next section shows an efficient way to code this information.

We now show how the SPIHT encoder uses these principles to progressively transmit the wavelet coefficients to the decoder (or write them on the compressed stream), starting with the most important information. We assume that a wavelet transform has already been applied to the image (SPIHT is a coding method, so it can work with any wavelet transform) and that the transformed coefficients $c_{i,j}$ are already stored in memory. The coefficients are sorted (ignoring their signs), and the sorting information is contained in an array m such that array element $m(k)$ contains the (i,j) coordinates of a coefficient $c_{i,j}$, and such that $|c_{m(k)}| \geq |c_{m(k+1)}|$ for all values of k. Table 25.30 lists hypothetical values of 16 coefficients. Each is shown as a 16-bit number where the most significant bit (bit 15) is the sign and the remaining 15 bits (numbered 14 through 0, top to bottom)

constitute the magnitude. The first coefficient $c_{m(1)} = c_{2,3}$ is $s1aci\ldots r$ (where s, a, etc., are bits). The second one $c_{m(2)} = c_{3,4}$ is $s1bdj\ldots s$, and so on.

k		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	sign	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
msb	14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	a	b	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	12	c	d	e	f	g	h	1	1	1	0	0	0	0	0	0	0
	11	i	j	k	l	m	n	o	p	q	1	0	0	0	0	0	0
	⋮	⋮	⋮	⋮													⋮
lsb	0	r	s	t	u	v	w	x	y		...		...		...		z
$m(k) = i, j$		2,3	3,4	3,2	4,4	1,2	3,1	3,3	4,2	4,1	...		...		...		4,3

Table 25.30: Transform Coefficients Ordered by Absolute Magnitudes.

The sorting information that the encoder has to transmit is the sequence $m(k)$, or

$$(2,3),\ (3,4),\ (3,2),\ (4,4),\ (1,2),\ (3,1),\ (3,3),\ (4,2),\ldots,(4,3).$$

In addition, it has to transmit the 16 signs, and the 16 coefficients in order of significant bits. A direct transmission would send the 16 numbers

$$ssssssssssssssss,\quad 1100000000000000,\quad ab11110000000000,$$
$$cdefgh1110000000,\quad ijklmnopq1000000,\ \ldots,\ rstuvwxy\ldots z,$$

but this is clearly wasteful. Instead, the encoder goes into a loop, where in each iteration it performs a *sorting step* and a *refinement step*. In the first iteration it transmits the number $l = 2$ (the number of coefficients $c_{i,j}$ in our example that satisfy $2^{14} \le |c_{i,j}| < 2^{15}$) followed by the two pairs of coordinates $(2,3)$ and $(3,4)$ and by the signs of the first two coefficients. This is done in the first sorting pass. This information enables the decoder to construct approximate versions of the 16 coefficients as follows: Coefficients $c_{2,3}$ and $c_{3,4}$ are constructed as the 16-bit numbers $s100\ldots0$. The remaining 14 coefficients are constructed as all zeros. This is how the most significant bits of the largest coefficients are transmitted to the decoder first.

The next step of the encoder is the refinement pass, but this is not performed in the first iteration.

In the second iteration the encoder performs both passes. In the sorting pass it transmits the number $l = 4$ (the number of coefficients $c_{i,j}$ in our example that satisfy $2^{13} \le |c_{i,j}| < 2^{14}$), followed by the four pairs of coordinates $(3,2)$, $(4,4)$, $(1,2)$, and $(3,1)$ and by the signs of the four coefficients. In the refinement step it transmits the two bits a and b. These are the 14th most significant bits of the two coefficients transmitted in the previous iteration.

The information received so far enables the decoder to improve the 16 approximate coefficients constructed in the previous iteration. The first six become

$$c_{2,3} = s1a0\ldots0,\ c_{3,4} = s1b0\ldots0,\ c_{3,2} = s0100\ldots0,$$
$$c_{4,4} = s0100\ldots0,\ c_{1,2} = s0100\ldots0,\ c_{3,1} = s0100\ldots0,$$

and the remaining 10 coefficients are not modified.

◇ **Exercise 25.10:** Perform the sorting and refinement passes of the next (third) iteration.

The main steps of the SPIHT encoder should now be easy to understand. They are as follows:

Step 1: Given an image to be compressed, perform its wavelet transform using any suitable wavelet filter, decompose it into transform coefficients $c_{i,j}$, and represent the resulting coefficients with a fixed number of bits. (In the discussion that follows we use the terms *pixel* and *coefficient* interchangeably.) We assume that the coefficients are represented as 16-bit signed-magnitude numbers. The leftmost bit is the sign, and the remaining 15 bits are the magnitude. (Notice that the sign-magnitude representation is different from the 2's complement method, which is used by computer hardware to represent signed numbers.) Such numbers can have values from $-(2^{15}-1)$ to $2^{15}-1$. Set n to $\lfloor \log_2 \max_{i,j}(c_{i,j}) \rfloor$. In our case n will be set to $\lfloor \log_2(2^{15}-1) \rfloor = 14$.
Step 2: Sorting pass: Transmit the number l of coefficients $c_{i,j}$ that satisfy $2^n \le |c_{i,j}| < 2^{n+1}$. Follow with the l pairs of coordinates and the l sign bits of those coefficients.
Step 3: Refinement pass: Transmit the nth most significant bit of all the coefficients satisfying $|c_{i,j}| \ge 2^{n+1}$. These are the coefficients that were selected in previous sorting passes (not including the immediately preceding sorting pass).
Step 4: Iterate: Decrement n by 1. If more iterations are needed (or desired), go back to step 2.

The last iteration is normally performed for $n = 0$, but the encoder can stop earlier, in which case the least important image information (some of the least significant bits of all the wavelet coefficients) will not be transmitted. This is the natural lossy option of SPIHT. It is equivalent to scalar quantization, but it produces better results than are usually achieved with scalar quantization, since the coefficients are transmitted in sorted order. An alternative is for the encoder to transmit the entire image (i.e., all the bits of all the wavelet coefficients) and the decoder can stop decoding when the reconstructed image reaches a certain quality. This quality can either be predetermined by the user or automatically determined by the decoder at run time.

25.4.1 Set Partitioning Sorting Algorithm

The method as described so far is simple, since we have assumed that the coefficients had been sorted before the loop started. In practice, the image may have $1\text{K} \times 1\text{K}$ pixels or more; there may be more than a million coefficients, so sorting all of them is too slow. Instead of sorting the coefficients, SPIHT uses the fact that sorting is done by comparing two elements at a time, and each comparison results in a simple yes/no result. Therefore, if both encoder and decoder use the same sorting algorithm, the encoder can simply send the decoder the sequence of yes/no results, and the decoder can use those

to duplicate the operations of the encoder. This is true not just for sorting but for any algorithm based on comparisons or on any type of branching.

The actual algorithm used by SPIHT is based on the realization that there is really no need to sort *all* the coefficients. The main task of the sorting pass in each iteration is to select those coefficients that satisfy $2^n \le |c_{i,j}| < 2^{n+1}$. This task is divided into two parts. For a given value of n, if a coefficient $c_{i,j}$ satisfies $|c_{i,j}| \ge 2^n$, then we say that it is *significant*; otherwise, it is called *insignificant*. In the first iteration, relatively few coefficients will be significant, but their number increases from iteration to iteration, because n keeps getting decremented. The sorting pass has to determine which of the significant coefficients satisfies $|c_{i,j}| < 2^{n+1}$ and transmit their coordinates to the decoder. This is an important part of the algorithm used by SPIHT.

The encoder partitions all the coefficients into a number of sets T_k and performs the significance test

$$\max_{(i,j)\in T_k} |c_{i,j}| \ge 2^n \ ?$$

on each set T_k. The result may be either "no" (all the coefficients in T_k are insignificant, so T_k itself is considered insignificant) or "yes" (some coefficients in T_k are significant, so T_k itself is significant). This result is transmitted to the decoder. If the result is "yes," then T_k is partitioned by both encoder and decoder, using the same rule, into subsets and the same significance test is performed on all the subsets. This partitioning is repeated until all the significant sets are reduced to size 1 (i.e., they contain one coefficient each, and that coefficient is significant). This is how the significant coefficients are identified by the sorting pass in each iteration.

The significance test performed on a set T can be summarized by

$$S_n(T) = \begin{cases} 1, & \max_{(i,j)\in T} |c_{i,j}| \ge 2^n, \\ 0, & \text{otherwise.} \end{cases} \tag{25.10}$$

The result, $S_n(T)$, is a single bit that is transmitted to the decoder. The result of each significance test becomes a single bit written on the compressed stream, which is why the number of tests should be minimized. To achieve this goal, the sets should be created and partitioned such that sets expected to be significant will be large and sets that are expected to be insignificant will contain just one element.

25.4.2 Spatial Orientation Trees

The sets T_k are created and partitioned using a special data structure called a *spatial orientation tree*. This structure is defined in a way that exploits the spatial relationships between the wavelet coefficients in the different levels of the subband pyramid. Experience has shown that the subbands in each level of the pyramid exhibit spatial similarity (Figure 25.8b). Any special features, such as a straight edge or a uniform region, are visible in all the levels at the same location.

The spatial orientation trees are illustrated in Figure 25.31a,b for a 16×16 image. The figure shows two levels, level 1 (the highpass) and level 2 (the lowpass). Each level is divided into four subbands. Subband LL2 (the lowpass subband) is divided into four groups of 2×2 coefficients each. Figure 25.31a shows the top-left group, and Figure 25.31b shows the bottom-right group. In each group, each of the four coefficients

(except the top-left one, marked in gray) becomes the root of a spatial orientation tree. The arrows show examples of how the various levels of these trees are related. The thick arrows indicate how each group of 4×4 coefficients in level 2 is the parent of four such groups in level 1. In general, a coefficient at location (i, j) in the image is the parent of the four coefficients at locations $(2i, 2j)$, $(2i + 1, 2j)$, $(2i, 2j + 1)$, and $(2i + 1, 2j + 1)$.

The roots of the spatial orientation trees of our example are located in subband LL2 (in general, they are located in the top-left LL subband, which can be of any size), but any wavelet coefficient, except the gray ones on level 1 (also except the leaves), can be considered the root of some spatial orientation subtree. The leaves of all those trees are located on level 1 of the subband pyramid.

In our example, subband LL2 is of size 4×4, so it is divided into four 2×2 groups, and three of the four coefficients of a group become roots of trees. Thus, the number of trees in our example is 12. In general, the number of trees is 3/4 the size of the highest LL subband.

Each of the 12 roots in subband LL2 in our example is the parent of four children located on the same level. However, the children of these children are located on level 1. This is true in general. The roots of the trees are located on the highest level, and their children are on the same level, but from then on, the four children of a coefficient on level k are themselves located on level $k - 1$.

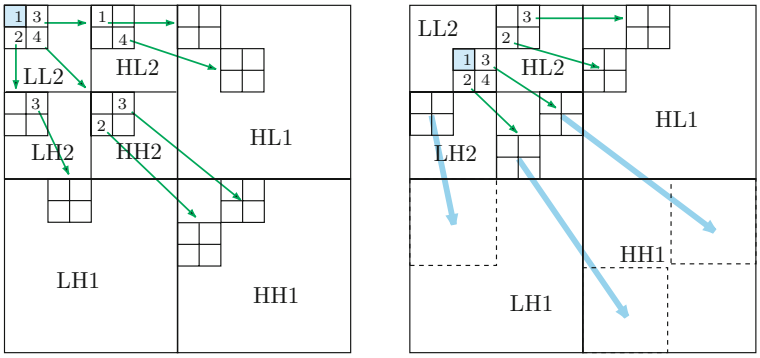


Figure 25.31: Spatial Orientation Trees in SPIHT.

We use the terms *offspring* for the four children of a node, and *descendants* for the children, grandchildren, and all their descendants. The set partitioning sorting algorithm uses the following four sets of coordinates:

1. $\mathcal{O}(i, j)$: the set of coordinates of the four offspring of node (i, j). If node (i, j) is a leaf of a spatial orientation tree, then $\mathcal{O}(i, j)$ is empty.
2. $\mathcal{D}(i, j)$: the set of coordinates of the descendants of node (i, j).
3. $\mathcal{H}(i, j)$: the set of coordinates of the roots of all the spatial orientation trees (3/4 of the wavelet coefficients in the highest LL subband).

4. $\mathcal{L}(i,j)$: The difference set $\mathcal{D}(i,j) - \mathcal{O}(i,j)$. This set contains all the descendants of tree node (i,j), except its four offspring.

The spatial orientation trees are used to create and partition the sets T_k. The set partitioning rules are as follows:

1. The initial sets are $\{(i,j)\}$ and $\mathcal{D}(i,j)$, for all $(i,j) \in \mathcal{H}$ (i.e., for all roots of the spatial orientation trees). In our example there are 12 roots, so there will initially be 24 sets: 12 sets, each containing the coordinates of one root, and 12 more sets, each with the coordinates of all the descendants of one root.
2. If set $\mathcal{D}(i,j)$ is significant, then it is partitioned into $\mathcal{L}(i,j)$ plus the four single-element sets with the four offspring of (i,j). In other words, if any of the descendants of node (i,j) is significant, then its four offspring become four new sets and all its other descendants become another set (to be significance tested in rule 3).
3. If $\mathcal{L}(i,j)$ is significant, then it is partitioned into the four sets $\mathcal{D}(k,l)$, where (k,l) are the offspring of (i,j).

Once the spatial orientation trees and the set partitioning rules are understood, the coding algorithm can be described.

25.4.3 SPIHT Coding

It is important to have the encoder and decoder test sets for significance in the same way. The coding algorithm therefore uses three lists called *list of significant pixels* (LSP), *list of insignificant pixels* (LIP), and *list of insignificant sets* (LIS). These are lists of coordinates (i,j) that in the LIP and LSP represent individual coefficients, and in the LIS represent either the set $\mathcal{D}(i,j)$ (a type A entry) or the set $\mathcal{L}(i,j)$ (a type B entry).

The LIP contains coordinates of coefficients that were insignificant in the previous sorting pass. In the current pass they are tested, and those that test significant are moved to the LSP. In a similar way, sets in the LIS are tested in sequential order, and when a set is found to be significant, it is removed from the LIS and is partitioned. The new subsets with more than one coefficient are placed back in the LIS, to be tested later, and the subsets with one element are tested and appended to the LIP or the LSP, depending on the results of the test. The refinement pass transmits the nth most significant bit of the entries in the LSP.

Figure 25.32 shows this algorithm in detail. Figure 25.33 is a simplified version, for readers who are intimidated by too many details.

The decoder executes the detailed algorithm of Figure 25.32. It always works in *lockstep* with the encoder, but the following notes shed more light on its operation:

1. Step 2.2 of the algorithm evaluates all the entries in the LIS. However, step 2.2.1 appends certain entries to the LIS (as type-B) and step 2.2.2 appends other entries to the LIS (as type-A). It is important to realize that all these entries are also evaluated by step 2.2 in the same iteration.
2. The value of n is decremented in each iteration, but there is no need to bring it all the way to zero. The loop can stop after any iteration, resulting in lossy compression. Normally, the user specifies the number of iterations, but it is also possible to have the user specify the acceptable amount of distortion (in units of MSE), and the encoder can use Equation (25.9) to decide when to stop the loop.

1. Initialization: Set n to $\lfloor \log_2 \max_{i,j}(c_{i,j}) \rfloor$ and transmit n. Set the LSP to empty. Set the LIP to the coordinates of all the roots $(i,j) \in \mathcal{H}$. Set the LIS to the coordinates of all the roots $(i,j) \in \mathcal{H}$ that have descendants.
2. Sorting pass:
   - 2.1 For each entry (i,j) in the LIP do:
     - 2.1.1 output $S_n(i,j)$;
     - 2.1.2 if $S_n(i,j) = 1$, move (i,j) to the LSP and output the sign of $c_{i,j}$;
   - 2.2 For each entry (i,j) in the LIS do:
     - 2.2.1 if the entry is of type A, then
       - output $S_n(\mathcal{D}(i,j))$;
       - if $S_n(\mathcal{D}(i,j)) = 1$, then
         - for each $(k,l) \in \mathcal{O}(i,j)$ do:
           - output $S_n(k,l)$;
           - if $S_n(k,l) = 1$, add (k,l) to the LSP, output the sign of $c_{k,l}$;
           - if $S_n(k,l) = 0$, append (k,l) to the LIP;
         - if $\mathcal{L}(i,j) \neq 0$, move (i,j) to the end of the LIS, as a type-B entry, and go to step 2.2.2; else, remove entry (i,j) from the LIS;
     - 2.2.2 if the entry is of type B, then
       - output $S_n(\mathcal{L}(i,j))$;
       - if $S_n(\mathcal{L}(i,j)) = 1$, then
         - append each $(k,l) \in \mathcal{O}(i,j)$ to the LIS as a type-A entry:
         - remove (i,j) from the LIS:
3. Refinement pass: for each entry (i,j) in the LSP, except those included in the last sorting pass (the one with the same n), output the nth most significant bit of $|c_{i,j}|$;
4. Loop: decrement n by 1 and go to step 2 if needed.

Figure 25.32: The SPIHT Coding Algorithm.

1. Set the threshold. Set LIP to all root nodes coefficients. Set LIS to all trees (assign type D to them). Set LSP to an empty set.
2. Sorting pass: Check the significance of all coefficients in LIP:
   - 2.1 If significant, output 1, output a sign bit, and move the coefficient to the LSP.
   - 2.2 If not significant, output 0.
3. Check the significance of all trees in the LIS according to the type of tree:
   - 3.1 For a tree of type D:
     - 3.1.1 if it is significant, output 1, and code its children:
       - if a child is significant, output 1, then a sign bit, add it to the LSP
       - if a child is insignificant, output 0 and add the child to the end of LIP.
       - if the children have descendants, move the tree to the end of LIS as type L, otherwise remove it from LIS.
     - 3.1.2 if it is insignificant, output 0.
   - 3.2 For a tree of type L:
     - 3.2.1 if it is significant, output 1, add each of the children to the end of LIS as an entry of type D and remove the parent tree from the LIS.
     - 3.2.2 if it is insignificant, output 0.
4. Loop: Decrement the threshold and go to step 2 if needed.

Figure 25.33: A Simplified SPIHT Coding Algorithm.

3. The encoder knows the values of the wavelet coefficients $c_{i,j}$ and uses them to compute bits S_n (Equation (25.10)), which it transmits (i.e., writes on the compressed stream). These bits are input by the decoder, which uses them to compute the values of $c_{i,j}$. The algorithm executed by the decoder is that of Figure 25.32 but with the word "output" changed to "input."
4. The sorting information, previously denoted by $m(k)$, is recovered when the coordinates of the significant coefficients are appended to the LSP in steps 2.1.2 and 2.2.1. This implies that the coefficients indicated by the coordinates in the LSP are sorted according to

$$\lfloor \log_2 |c_{m(k)}| \rfloor \geq \lfloor \log_2 |c_{m(k+1)}| \rfloor,$$

for all values of k. The decoder recovers the ordering because its three lists (LIS, LIP, and LSP) are updated in the same way as those of the encoder (remember that the decoder works in lockstep with the encoder). When the decoder inputs data, its three lists are identical to those of the encoder at the moment it (the encoder) output that data.
5. The encoder starts with the wavelet coefficients $c_{i,j}$; it never gets to "see" the actual image. The decoder, however, has to display the image and update the display in each iteration. In each iteration, when the coordinates (i, j) of a coefficient $c_{i,j}$ are moved to the LSP as an entry, it is known (to both encoder and decoder) that $2^n \leq |c_{i,j}| < 2^{n+1}$. As a result, the best value that the decoder can give the coefficient $\hat{c}_{i,j}$ that is being reconstructed is midway between 2^n and $2^{n+1} = 2 \times 2^n$. Thus, the decoder sets $\hat{c}_{i,j} = \pm 1.5 \times 2^n$ (the sign of $\hat{c}_{i,j}$ is input by the decoder just after the insertion). During the refinement pass, when the decoder inputs the actual value of the nth bit of $c_{i,j}$, it improves the value 1.5×2^n by either adding 2^{n-1} to it (if the input bit was a 1) or subtracting 2^{n-1} from it (if the input bit was a 0). This way, the decoder can improve the appearance of the image (or, equivalently, reduce the distortion) during *both* the sorting and refinement passes.

It is possible to improve the performance of SPIHT by entropy coding the encoder's output, but experience shows that the added compression gained in this way is minimal and does not justify the additional expense of both encoding and decoding time. It turns out that the signs and the individual bits of coefficients output in each iteration are uniformly distributed, so entropy coding them does not produce any compression. The bits $S_n(i, j)$ and $S_n(\mathcal{D}(i, j))$, on the other hand, are distributed nonuniformly and may gain from such coding.

25.4.4 Example

We assume that a 4×4 image has already been transformed, and the 16 coefficients are stored in memory as 6-bit signed-magnitude numbers (one sign bit followed by five magnitude bits). They are shown in Figure 25.34, together with the single spatial orientation tree. The coding algorithm initializes LIP to the one-element set $\{(1, 1)\}$, the LIS to the set $\{\mathcal{D}(1, 1)\}$, and the LSP to the empty set. The largest coefficient is 18, so n is set to $\lfloor \log_2 18 \rfloor = 4$. The first two iterations are shown.
Sorting Pass 1:
$2^n = 16$.
Is (1,1) significant? yes: output a 1.
LSP = $\{(1, 1)\}$, output the sign bit: 0.

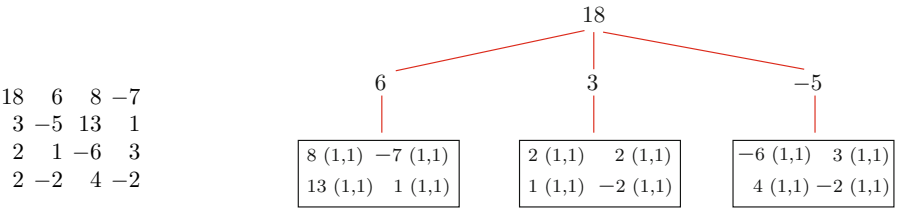


Figure 25.34: Sixteen Coefficients and One Spatial Orientation Tree.

Is $\mathcal{D}(1,1)$ significant? no: output a 0.
LSP = $\{(1,1)\}$, LIP = $\{\}$, LIS = $\{\mathcal{D}(1,1)\}$.
Three bits output.
Refinement pass 1: no bits are output (this pass deals with coefficients from sorting pass $n-1$).
Decrement n to 3.
Sorting Pass 2:
$2^n = 8$.
Is $\mathcal{D}(1,1)$ significant? yes: output a 1.
Is $(1,2)$ significant? no: output a 0.
Is $(2,1)$ significant? no: output a 0.
Is $(2,2)$ significant? no: output a 0.
LIP = $\{(1,2),(2,1),(2,2)\}$, LIS = $\{\mathcal{L}(1,1)\}$.
Is $\mathcal{L}(1,1)$ significant? yes: output a 1.
LIS = $\{\mathcal{D}(1,2),\mathcal{D}(2,1),\mathcal{D}(2,2)\}$.
Is $\mathcal{D}(1,2)$ significant? yes: output a 1.
Is $(1,3)$ significant? yes: output a 1.
LSP = $\{(1,1),(1,3)\}$, output sign bit: 1.
Is $(2,3)$ significant? yes: output a 1.
LSP = $\{(1,1),(1,3),(2,3)\}$, output sign bit: 1.
Is $(1,4)$ significant? no: output a 0.
Is $(2,4)$ significant? no: output a 0.
LIP = $\{(1,2),(2,1),(2,2),(1,4),(2,4)\}$,
LIS = $\{\mathcal{D}(2,1),\mathcal{D}(2,2)\}$.
Is $\mathcal{D}(2,1)$ significant? no: output a 0.
Is $\mathcal{D}(2,2)$ significant? no: output a 0.
LIP = $\{(1,2),(2,1),(2,2),(1,4),(2,4)\}$,
LIS = $\{\mathcal{D}(2,1),\mathcal{D}(2,2)\}$,
LSP = $\{(1,1),(1,3),(2,3)\}$.
Fourteen bits output.
Refinement pass 2: After iteration 1, the LSP included entry $(1,1)$, whose value is $18 = 10010_2$.
One bit is output.
Sorting Pass 3:
$2^n = 4$.

Is $(1,2)$ significant? yes: output a 1.
LSP $= \{(1,1),(1,3),(2,3),(1,2)\}$, output a sign bit: 1.
Is $(2,1)$ significant? no: output a 0.
Is $(2,2)$ significant? yes: output a 1.
LSP $= \{(1,1),(1,3),(2,3),(1,2),(2,2)\}$, output a sign bit: 0.
Is $(1,4)$ significant? yes: output a 1.
LSP $= \{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4)\}$, output a sign bit: 1.
Is $(2,4)$ significant? no: output a 0.
LIP $= \{(2,1),(2,4)\}$.
Is $D(2,1)$ significant? no: output a 0.
Is $D(2,2)$ significant? yes: output a 1.
Is $(3,3)$ significant? yes: output a 1.
LSP $= \{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3)\}$, output a sign bit: 0.
Is $(4,3)$ significant? yes: output a 1.
LSP $= \{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3),(4,3)\}$, output a sign bit: 1.
Is $(3,4)$ significant? no: output a 0.
LIP $= \{(2,1),(2,4),(3,4)\}$.
Is $(4,4)$ significant? no: output a 0.
LIP $= \{(2,1),(2,4),(3,4),(4,4)\}$.
LIP $= \{(2,1),(3,4),(3,4),(4,4)\}$, LIS $= \{\mathcal{D}(2,1)\}$,
LSP $= \{(1,1),(1,3),(2,3),(1,2),(2,2),(1,4),(3,3),(4,3)\}$.
Sixteen bits output.
Refinement Pass 3:
After iteration 2, the LSP included entries $(1,1)$, $(1,3)$, and $(2,3)$, whose values are $18 = 10010_2$, $8 = 1000_2$, and $13 = 1101_2$. Three bits are output
After two iterations, a total of 37 bits has been output.

25.5 QTCQ

Closely related to SPIHT, the QTCQ (quadtree classification and trellis coding quantization) method [Banister and Fischer 99] employs fewer lists than SPIHT and explicitly forms classes of the wavelet coefficients for later quantization by means of the ACTCQ and TCQ (arithmetic and trellis coded quantization) algorithms of [Joshi, Crump, and Fischer 93].

The method uses the spatial orientation trees originally developed by SPIHT. This type of tree is a special case of a quadtree (Section 18.7). The encoding algorithm is iterative. In the nth iteration, if any element of this quadtree is found to be significant, then the four highest elements in the tree are defined to be in class n. They also become roots of four new quadtrees. Each of the four new trees is tested for significance, moving down each tree until all the significant elements are found. All the wavelet coefficients declared to be in class n are stored in a *list of pixels* (LP). The LP is initialized with all the wavelet coefficients in the lowest frequency subband (LFS). The test for significance is performed by the function $S_T(k)$, which is defined by

$$S_T(k) = \begin{cases} 1, & \max_{(i,j)\in k} |C_{i,j}| \geq T, \\ 0, & \text{otherwise}, \end{cases}$$

where T is the current threshold for significance and k is a tree of wavelet coefficients. The QTCQ encoding algorithm uses this test, and is listed in Figure 25.35.

The QTCQ decoder is similar. All the outputs in Figure 25.35 should be replaced by inputs, and ACTCQ encoding should be replaced by ACTCQ decoding.

```
1. Initialization:
   Initialize LP with all C_{i,j} in LFS,
   Initialize LIS with all parent nodes,
   Output n = ⌊log2(max|C_{i,j}|/q)⌋.
   Set the threshold T = q2^n, where q is a quality factor.
2. Sorting:
   for each node k in LIS do
     output S_T(k)
     if S_T(k) = 1 then
      for each child of k do
       move coefficients to LP
       add to LIS as a new node
      endfor
      remove k from LIS
     endif
   endfor
3. Quantization: For each element in LP,
    quantize and encode using ACTCQ.
    (use TCQ step size Δ = α·q).
4. Update: Remove all elements in LP. Set T = T/2. Go to step 2.
```

Figure 25.35: QTCQ Encoding.

The QTCQ implementation, as described in [Banister and Fischer 99], does not transmit the image progressively, but the authors claim that this property can be added to it.

How lovely the little river is, with its dark changing wavelets.
—George Eliot, *The Mill on the Floss* (1860)

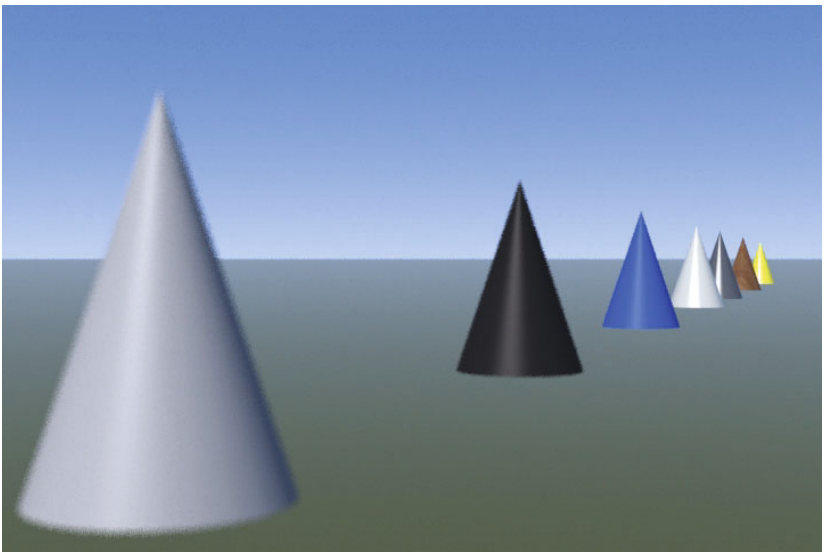


Plate P.1. Illustrating Depth of Field (Modo).

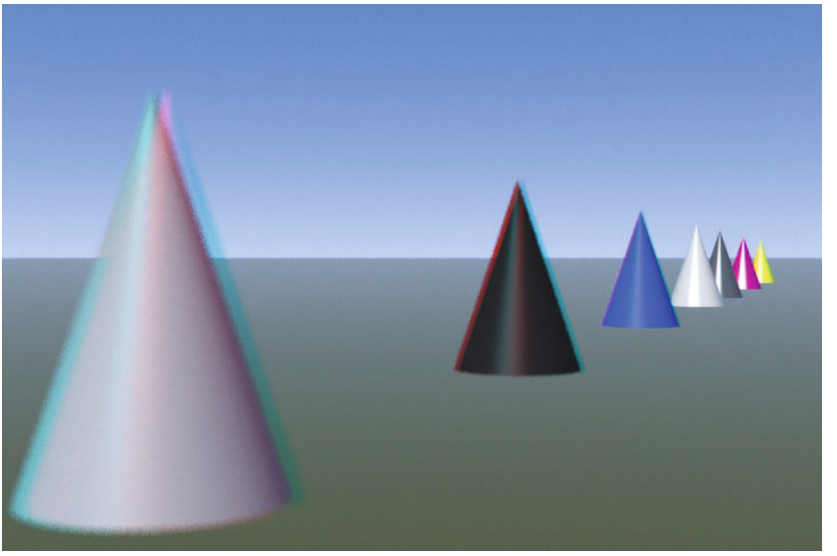
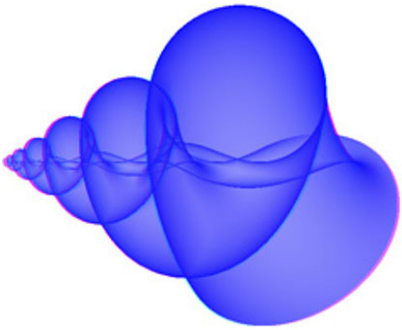
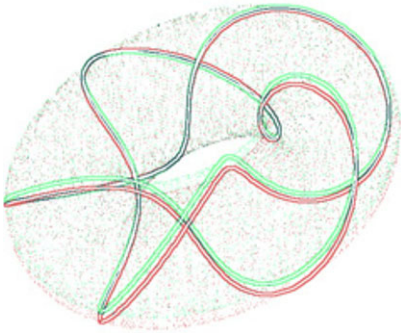


Plate P.2. A Stereo Image (Modo and Photoshop).

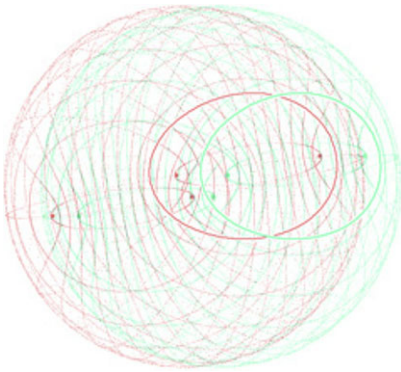


Snail Shell

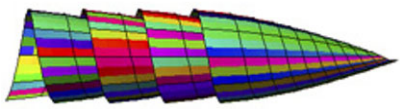


Torus Knot

Hopf Fibered Linked Tori



Spherical Ellipse

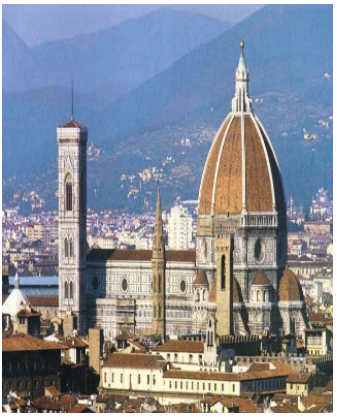


Helicoid

Plate P.3. Mathematical Objects, Mostly in Stereo (3D-ExplorMath).

Original

Reflected Horizontally



Scaled Vertically

Halftone

Sheared

Rotated

Scaled Horizontally

Plate Q.1. Affine Transformations (Photoshop).

Plate Q.2. Kyoto Train Station (Left) and Teramachi Arcade (Right), Courtesy of Ari Salomon.

Plate R.1. Kyoto Train Station (Individual Parts), Courtesy of Ari Salomon.

Plate R.2. Free-Form Bitmap Deformations (PlasticBeauty).